FOR FURTHER TRAN

ARPA ORDER NO. 2223

ISI/RR-77-65
November 1977

AD A055501

**Mark S. Moriconi**
*University of Texas at Austin*
and USC/Information Sciences Institute

# A System for Incrementally Designing and Verifying Programs
## Volume 1

DDC
JUN 21 1978
F

AD No.
DDC FILE COPY

This document has been approved for public release and sale; its distribution is unlimited.

*INFORMATION SCIENCES INSTITUTE*
*UNIVERSITY OF SOUTHERN CALIFORNIA*
*4676 Admiralty Way/ Marina del Rey/ California 90291*
*(213) 822-1511*

78 06 19 058

Mark S. Moriconi
A System for Incrementally Designing and Verifying Programs Volume 1
ISI/RR-77-65

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

(14)

## REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ISI/RR-77-65
2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER

(6) 4. TITLE *(and Subtitle)*: A System for Incrementally Designing and Verifying Programs, Volume I.

(9) 5. TYPE OF REPORT & PERIOD COVERED: Research Rept.

6. PERFORMING ORG. REPORT NUMBER

7. AUTHOR(s): (10) Mark S. Moriconi

(15) 8. CONTRACT OR GRANT NUMBER(s): DAHC-15-72-C-0308, ARPA Order - 2223

9. PERFORMING ORGANIZATION NAME AND ADDRESS
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90291

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Defense Advanced Research Projects Agency
1400 Wilson Blvd.
Arlington, VA 22209

(11) 12. REPORT DATE: Jan[redacted] 1978

(12) 13. NUMBER OF PAGES: ~~117~~ 124 P.

14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*

15. SECURITY CLASS. *(of this report)*: Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

This report is approved for public release and sale; distribution is unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

incremental, interactive dialog, interactive system, network, program design, program verification, sorting, specification

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

(Over)

DD FORM 1 JAN 73 1473 — EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

447952

78 06 19 058 hc

20. ABSTRACT

SID (System for Incremental Development) is a computer system for incrementally designing and verifying large, complex programs. It executes commands, proposes actions, answers questions, and accepts and reasons about new or changed information. SID has three main, distinctive characteristics: (1) it provides several useful incremental capabilities, including the ability to respond to changes by ensuring that the final problem solution is consistent and by keeping intact still-valid work without complete reprocessing; (2) its user interface has the ability to guide the user through the design and verification and to engage in an interactive English dialog about the potential effects of changes; (3) it supports a substantial programming language which includes features for generating run-time checks, stating concurrent processes and shared data, and developing data abstractions. SID has been used to completely design and verify several programs.

Volume 2 (appendix) contains a transcript of a session with SID in which a simple message switching network that allows secure, asynchronous message transfer among a fixed number of users is incrementally developed.

1473B

*ARPA ORDER NO. 2223*

*ISI/RR-77-65*
*November 1977*

**Mark S. Moriconi**
University of Texas at Austin
and USC/Information Sciences Institute

# A System for Incrementally Designing and Verifying Programs
# Volume 1

*INFORMATION SCIENCES INSTITUTE*

*UNIVERSITY OF SOUTHERN CALIFORNIA*

*4676 Admiralty Way/ Marina del Rey/ California 90291*
*(213) 822-1511*

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308, ARPA ORDER NO. 2223.

VIEWS AND CONCLUSIONS CONTAINED IN THIS STUDY ARE THE AUTHOR'S AND SHOULD NOT BE INTERPRETED AS REPRESENTING THE OFFICIAL OPINION OR POLICY OF ARPA, THE U.S. GOVERNMENT OR ANY OTHER PERSON OR AGENCY CONNECTED WITH THEM.

THIS DOCUMENT APPROVED FOR PUBLIC RELEASE AND SALE: DISTRIBUTION IS UNLIMITED.

# ABSTRACT

SID (System for Incremental Development) is a computer system for incrementally designing and verifying large, complex programs. It executes commands, proposes actions, answers questions, and accepts and reasons about new or changed information.

SID has three main, distinctive characteristics. First, it provides several useful incremental capabilities, including the ability to respond to changes by ensuring that the final problem solution is consistent and by keeping intact still-valid work without complete reprocessing. Second, its user interface has the ability to guide the user through the design and verification and to engage in an interactive English dialog about the potential effects of changes. It not only previews these effects, but also explains the reasons for them. Third, it supports a substantial programming language, which includes features for generating run-time checks, stating concurrent processes and shared data, and developing data abstractions. Complete type checking is performed on programs, specifications, and assumed properties.

The system responds to changes by applying a new methodology that, although couched here in the domain of design and verification, may also be useful in other areas in which incremental reaction to change is important. Parts of a general framework for designing incremental systems are proposed.

SID has been used to completely design and verify several programs, including a simple message switching network that allows secure, asynchronous message transfer among a fixed number of users.

ACCESSION for
NTIS White Section
DDC Buff Section
UNANNOUNCED
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. SPECIAL
A1

# ACKNOWLEDGEMENTS

I would like to express my appreciation to my thesis advisor, W. W. Bledsoe, for providing just the right amount of encouragement and guidance and also to the other members of the thesis committee: Donald Good, Ralph London, Norman Martin, and Raymond Yeh.

The design and implementation of the system was a collective effort of several people whose specific contributions are noted in the text. I would like to offer special thanks to Richard Cohen and Mabry Tyson, without whose diligence and expertise neither the system nor the reported examples would have been finished.

Several people helped by offering suggestions about content or style. Key ideas on how to respond to changes grew out of conversations with David Wile. The notion of providing an explanation facility for previewing the effects of changes was suggested by W. W. Bledsoe. Donald Good provided general technical guidance and suggested many stylistic improvements. Others who helped are: Mike Ballantyne, John Heafner, Dallas Lankford, Donald Lynn, David Musser, David Wilczynski, and Marty Yonke.

Nancy Bryan provided helpful editorial comments on a previous draft. Parts of this document were typed by Kathie Richardson. Drawings were done by Nelson Lucas. Raymond Bates helped with the formatting.

This research was supported at The University of Texas at Austin by the Certifiable Minicomputer Project and by the National Science Foundation (Grant No. MCS74-12866) and at USC Information Sciences Institute by the Defense Advanced Research Projects Agency (Contract No. DAHC 15-72-C-0308).

# CONTENTS

# CHAPTER 1

# INTRODUCTION

## 1.1 Incremental Program Design and Verification

When designing and verifying programs, a series of steps is performed in which the data is not only repeatedly added to, but also often revised due to conceptual reformulation or error correction. This evolutionary, or incremental, process entails using knowledge of design and verification and of the context of revisions. To effectively support this kind of activity with a computer, a program with a highly integrated, incremental view of design and verification is needed.

The objective of this research is to provide such a system for incrementally designing and verifying programs of significant size and complexity. Consequently, three main system design goals emerged. The system should:

- Provide useful incremental capabilities. This includes the ability to respond to changes by ensuring that the final problem solution is consistent and by keeping intact still-valid work without complete reprocessing.

- Provide a good user interface. This includes the ability to guide the user through the design and verification and to answer questions about it.

- Support a substantial programming language. Desirable language features include extensive programming and specification facilities, verifiability, and constructs for assisting in incremental development.

---

This report reproduces a thesis of the same title submitted to the Department of Computer Sciences at The University of Texas at Austin on 6 December 1977 in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Author's current address: Computer Science Laboratory, Stanford Research Institute, Menlo Park, California 94025

This dissertation describes a working prototype system -- called SID (System for Incremental Development) -- that realizes these goals. It is capable of accepting and reasoning about new or changed information, of carrying out manipulations that are too complex, cumbersome, and tedious to do reliably by hand, of dynamically proposing actions to the user, and of engaging in an English dialog about the task. It allows the user to design his program by any of several strategies, verify it in parallel or any desired order, and make changes whenever convenient.

SID supports the language Gypsy [Ambler, et al. 77], whose consistent integration of both programming and specification statements allows the system, among other things, to type check programs, specifications, and assumed properties. Gypsy includes features for writing several forms of specifications, generating run-time checks, stating concurrent processes and shared data, handling errors, developing data abstractions, and assisting in program development.

SID responds to changes by applying a new methodology that, although couched here in the context of design and verification, may also be useful in other areas in which incremental reaction to change is important. Consider, for example, the development of new mathematical theories. It is commonplace to introduce lemmas or definitions, revise them (or previous ones) for any of several reasons, then try proofs again. As a practical matter, these changes should be compensated for without completely redoing previous work that remains valid.

The organization of this dissertation is as follows. Section 1.2 presents a detailed sample session which illustrates the functioning of the system, with emphasis on features that enhance incremental development. The example developed is referred to in later chapters.

Chapter 2 compares SID with several other program verification systems. Differences in philosophy, capabilities, and achievements are described.

Chapter 3 describes the system -- its design, user interface, incremental capabilities, and functional components. A multi-level overview of the design is given and the impact of the incremental approach is studied. The user interface is described, the novelty of which lies in its dual-mode philosophy, giving the user the option of directing the system by giving commands, or being guided through the system by accepting suggestions, or a combination of both. Suggestions are based on the current state of development. Also of practical importance is its ability to answer "what" and "why" questions about the potential effects of changes. The incremental capabilities of the system are itemized, accompanied by an explanation of how those that are not covered in Chapter 4 are accomplished. Characteristics of each functional component, some of which were developed by others specifically for this system, are described and illustrated. Excerpts from the scenario of Section 1.2 are used in explanations.

Chapter 4 focuses on the methodology for responding to changes to programs, specifications, and properties assumed in proofs. The effects of changes are determined by exploiting useful constraints on how objects in the domain interact with one another. For example, previous verification work is kept intact by employing logical rules which describe *when changes to specifications have no effect.*

Practical considerations significantly influenced the design of the methodology. For example, the methodology allows temporary inconsistencies (such as references to programs to be defined later), applies to any of several design and verification strategies, and adapts to a class of Pascal-like languages and their related proof methods. The description of the methodology given in Chapter 4 is intended to be formal enough to allow implementations to be direct, while suppressing unimportant details.

Chapter 5 studies the problem of designing other incremental systems. Key issues, practical tradeoffs, and some problems are discussed. A general framework is proposed for answering some of the questions raised. Other questions are explored by means of illustrative examples.

## 1.2 Example Session

This section contains an example session with SID in which a sorting program is incrementally designed and verified using a top-down strategy. This session illustrates some ways in which SID executes user commands, proposes actions, answers questions about aspects of the design and verification, and accepts and reasons about new or changed information throughout the development.

In addition to this sorting example, a message switching network that allows secure, asynchronous message transfer among a fixed number of users is developed in Appendix A. The network, being larger and more complex, underscores the importance of the system's incremental philosophy. Some of the interesting features of the network are concurrency, data abstraction, and error handling.

The session given below is an abstracted version of an actual session that is intended to convey a representative sample of SID's incremental capabilities. The sorting program developed consists of three executable programs (Exchange_sort being the top-level program which invokes Location_of_max and Exchange) and their accompanying data declarations and specifications. The scenario shown has several key stages.

1. ***Initial design***. Exchange_sort, which is only partially defined and contains unresolved external references, is entered into the system. When

attempting to generate verification conditions (abbreviated as VCs), the system figures out that the operations cannot be performed due to constraints imposed by inconsistencies.

2. ***Removal of inconsistencies.*** The system accepts some new information, analyzing it within and incorporating it into the existing context.

3. ***Verification.*** Determining that VCs can now be generated, SID generates them for those paths in Exchange_sort that are completely defined, and temporarily ignores the paths that are not. VCs contain only references to specifications of called programs, instead of the specifications themselves. Each VC is then proved, expanding complete specifications, or parts of specifications, and extending the collection of facts known to the system as needed during proofs.

4. ***Continue design.*** The user and system engage in a conversation in which SID previews and explains the potential effects of intended changes. Guided by this interchange, the user makes several changes -- the definition of Exchange_sort is completed, previously-entered information is changed (e.g., some incomplete specifications are extended), new information is introduced (e.g., Location_of_max and Exchange are partially defined). The system fits all these changes into the existing problem structure.

5. ***Continue verification.*** The system not only generates the new VCs for Exchange_sort, but also keeps intact its still-valid proofs. Proof of the new VCs and iterations through (4) and (5) for Location_of_max and Exchange are omitted.

6. ***Maintenance.*** The session resumes after the design and verification is completed to demonstrate the utility of SID's explanation facility as a maintenance tool. An illustrative dialog explores the effects of a varied sample of possible revisions and how the system would react to them.

It should be emphasized that the purpose of this example is not to convey specific details of the sorting program being developed, but rather to illustrate the kinds of things SID does to enhance incremental design and verification. The reader is encouraged to at least browse through the entire example to get an understanding of the overall scenario. The dialog at the end of the session is particularly important in that it provides a brief capsule of SID's effectiveness in handling changes.

Below, lines typed by a person are underlined to distinguish them from the computer's response. The text is taken verbatim from the actual session, and is all inclusive except where explicitely marked. Italicized lines are comments describing some of the features being exhibited.

```
Exec-> ?
Commands are:  Add, Delete, EDit, EXplain, Help, PRint, PROve,
          Restore, SAve, SUggest, Translate, Vcs, ?E, ?, ??.
Exec-> T

  Enter file name-> File1.Srt
```

*Typing a carriage return before completing a multi-word command causes the system to prompt for remaining parts. The design and verification begins by reading the file File1.Srt into the system. The file contains programs and specifications which are echoed by the system as they are read.*

```
const n:int = pending;

type int_array = array ([1..n]) of int;

function Exchange_sort(A:int_array):int_array =
begin
  entry N ge 1;
  exit (all I:int,
        I in [1..N] ->  Exchange_sort(A)[I]
                     = Value_of_max(Exchange_sort(A),1,I))
      and IsPerm(A, Exchange_sort(A));
  var B:int_array := A;
  var K:int := N;
  keep  K in [1..N];
  loop
    assert (all I:int, I in [k+1..N] -> B[I] = Value_of_max(B,1,I))
      and K in [1..N]  and  IsPerm(A,B);
    if K = 1 then leave
    else pending end;
  end;
  result := B;
end;
```

No syntax errors detected.

Semantic errors detected:

```
function EXCHANGE_SORT(A : INT_ARRAY) : INT_ARRAY =
begin
  entry N ge 1;
  exit     (all I : INT,    I in [1..N]
                    ->   EXCHANGE_SORT(A)[I]
                      = VALUE_OF_MAX[EXCHANGE_SORT[A], 1, I])
                                 ↑P1
      and ISPERM[A, EXCHANGE_SORT[A]];
            ↑P1
      .
      .
      .

end;
```

P1:  NAME NOT DECLARED

*This specifies and partially defines the sorting program Exchange_sort. Its exit assertion states that the output array is sorted into ascending order and consists of all the elements of the input array, and nothing else. The keep assertion states that the array index K is always within range. The "pending" is used to postpone parts of the implementation. Semantic errors, detected in both specifications and executable code, are due to unresolved external references. Although incomplete and semantically incorrect, Exchange_sort is stored for later use.*

Exec-> VCS Exchange_sort

**** The operation VCS is not legitimate because EXCHANGE_SORT is semantically incorrect.

*Knowledge of specific tasks is used to determine if a particular operation can be performed, temporarily cannot be performed, or cannot ever be performed.*

Exec-> Print Status All

The current design and verification status is:
  Waiting for VC generation (semantics must be corrected): EXCHANGE_SORT
  Constants/Types: INT_ARRAY, N

*Status summaries reflecting the current state of development aid in keeping track of progress and context. This summary of the initial design indicates that semantic errors in Exchange_sort must be corrected before generating its VCs, and that Int_array and N are data declarations. The user proceeds to correct these errors by changing the present design.*

```
Exec-> Edit
```

*A standard text editor is used for creating and changing programs and specifications. The user leaves the system, performs the desired editing, then returns to the system in the same state in which he left it.*

```
Exec-> Translate File2.Srt

function Value_of_max(A:int_array; I,J:int):int = begin end;

function IsPerm(X,Y:int_array) : boolean =
begin
  exit (assume
          (all Z:int_array, IsPerm(Z,Z)));
end;
```

*All data entering the system, whether new or changed, is analyzed within the current context, then fit into it forming a new problem state. For example, SID uses the previously-defined type Int_array in type checking Value_of_max and IsPerm. Both of these new functions have null bodies (as opposed to pending ones), indicating that they are solely for specification purposes and will not be implemented.*

```
  No syntax errors detected.
  No semantic errors detected.

Exec-> Print Status All

The current design and verification status is:
   Waiting for VC generation (must check semantics): EXCHANGE_SORT
   For specifications only: ISPERM, VALUE_OF_MAX
   Constants/Types: INT_ARRAY, N
```

*The system knows that semantic errors previously detected in Exchange_sort may now be corrected. It indicates this by changing "semantics must be corrected" in the previous status summary to "must check semantics" in this one.*

Exec-> VCS Exchange_sort

*The system determines that this command can now be carried out and displays both program paths and their accompanying verification conditions.*

```
Generating VCs for FUNCTION EXCHANGE_SORT

Found 1-ST path
Found 2-ND path
Found 3-RD path
1 pending path encountered.

-----------------------------------------------
Beginning new path...
B := A
K := N

Assume (unit entry condition)
    N ge 1

Continuing in path...

Assume (KEEP assertion)
    K in [1..N]

Entering loop...

Evaluating  VALUE_OF_MAX(B, 1, I-1)
```

*Entry assertions of programs called from executable code or specifications are proved at all calling sites. The system evaluates the entry assertion of Value_of_max on the arguments shown. The resulting VC is automatically reduced to true and is therefore suppressed. Had it not reduced to true, a VC would have been spawned.*

```
Continuing in path...

Evaluating  ISPERM(A, B)
Continuing in path...
```

```
ASSERT     (all I#1 : INT,    I#1 in [K+1..N]
                    -> B[I#1] = VALUE_OF_MAX(B, 1, I#1))
      and K in [1..N]
      and ISPERM(A, B)
- - - - - - - - - - - - - - - - - - - - - - - -
Must verify ASSERT condition
Verification condition EXCHANGE_SORT#1
 H1: 1 le N
 -->
 C1: ISPERM(A, A)
```

*Notice that IsPerm is referenced but that specifications from IsPerm were not added to the hypothesis. Instead, such specifications are added completely or partially when needed during proofs. This policy, together with the one of simplifying during generation, limits the size of VCs.*

```
- - - - - - - - - - - - - - - - - - - - - - - -
End of path
-----------------------------------------------


-----------------------------------------------
Beginning new path...
Continuing in LOOP ...

Assume (from last assertion)
       (all I#1 : INT,    I#1 in [K+1..N]
                    -> B[I#1] = VALUE_OF_MAX(B, 1, I#1))
    and K in [1..N]
    and ISPERM(A, B)

Assume (KEEP assertion)
    K in [1..N]

Assume (IF test failed)
    (not K=1)

Entering PENDING statement...cannot continue in this path

End of path
-----------------------------------------------
```

---

**Beginning new path...**

.
.
.

**End of path**

---

**Exec-> Print Status $**

*Using "$" in a command (i.e., typing the escape key) is a way of having the system make default assignments. The assigned value here is Exchange_sort.*

**Waiting for pending path to be filled in.**
**EXCHANGE_SORT#1 . . . EXCHANGE_SORT#3 waiting to be proved.**

*Presently, Exchange_sort contains a pending path that eventually must be implemented and has three VCs that must be proved.*

**Exec-> Suggest**

*Rather than continuing to direct the system by giving commands, the user wants SID to propose actions needed to obtain a problem solution. Acceptance of a suggested action is indicated by typing "$".*

**Suggest proving VC called EXCHANGE_SORT#1-> $**
**Entering Prover with verification condition EXCHANGE_SORT#1**
**H1: I le N**
**-->**
**C1: ISPERM(A, A)**

.
.
.

**Prover-> Use IsPerm**

.
.
.

**EXCHANGE_SORT#1 proved in theorem prover.**

*Proofs are omitted for brevity; only interactive proof steps which bring additional knowledge to bear on the deduction are shown. The command Use IsPerm adds the entire exit assertion of IsPerm to the current hypothesis.*

**Suggest proving VC called EXCHANGE_SORT#2-> Print Status Exchange_sort**

*Instead of accepting a suggestion, the user can issue a command. If command execution causes a state change, the user is returned to command mode under the assumption that he wants to pursue a line of reasoning different from the scenario now modelled internally. For commands that do not change state (e.g., print commands), the same suggestion is repeated after executing the command.*

**Waiting for pending path to be filled in.**
**Waiting to be proved: EXCHANGE_SORT#2, EXCHANGE_SORT#3**
**Proved in theorem prover: EXCHANGE_SORT#1**

**Suggest proving VC called EXCHANGE_SORT#2-> §**
**Entering Prover with verification condition EXCHANGE_SORT#2**
**H1: all I#1 : INT, I#1 in [K+1..N] -> VALUE_OF_MAX(B, 1, I#1) = B[I#1]**
**H2: K=1**
**H3: ISPERM(A, B)**
**H4: K in [1..N]**
**-->**
**C1: 1 le N**
.
.
.

**EXCHANGE_SORT#2 proved in theorem prover.**

**Suggest proving VC called EXCHANGE_SORT#3-> §**
**Entering Prover with verification condition EXCHANGE_SORT#3**
**H1: all I#1 : INT, I#1 in [K+1..N] -> VALUE_OF_MAX(B, 1, I#1) = B[I#1]**
**H2: K=1**
**H3: ISPERM(A, B)**
**H4: K in [1..N]**
**-->**
**C1: all I : INT, I in [1..N] -> VALUE_OF_MAX(B, 1, I) = B[I]**
.
.
.

Prover-> Use Lemma
Enter lemma . . .
* all L:int, all P:Int_Array,
*  L in [1..N] -> Value_of_max(P,L,L) = P[L];
Lemma added . . . Its name is LEMMA#1

*New lemmas (and also rewrite rules) are added when needed during proofs. They are parsed, type checked within the context of the overall problem, and named for subsequent reference. "*" is the prompting symbol of the translator.*

.
.
.

EXCHANGE_SORT#3 proved in theorem prover.

Suggest fully defining EXCHANGE_SORT-> Print Status All

The current design and verification status is:
  Waiting for pending path to be filled in: EXCHANGE_SORT
  Proved: EXCHANGE_SORT
  For specifications only: ISPERM, VALUE_OF_MAX
  Constants/Types: INT_ARRAY, N

*Since all VCs for Exchange_sort are proved, the user now is ready to continue the design. But before actually changing the present design, he uses SID to explore the potential effects of some intended changes.*

Suggest fully defining EXCHANGE_SORT-> Explain

<- What are the effects of fully defining Exchange_sort?

EXCHANGE_SORT may have additional VCs.

<- What are the effects of changing the exit assertion of Value_of_max?

No effects on VALUE_OF_MAX. The verification of other programs is not affected.

<- Why?

*The system remembers immediately preceeding sentences to answer questions like this one.*

No effects on VALUE_OF_MAX because it is for specifications only and does not have VCs. No external effects because properties from VALUE_OF_MAX have not been used in proving any VCs.

<- What are the effects of changing the exit of IsPerm?

No effects on ISPERM. The verification of other programs is not affected if the formula

Changed exit specification
-->
C1: all Z : INT_ARRAY, ISPERM(Z, Z)

is true. If not, the change invalidates the verification of EXCHANGE_SORT.

<- Why?

No effects on ISPERM because it is for specifications only and does not have VCs. Invalid verification because properties from ISPERM were used in proving EXCHANGE_SORT#1.

*The system gives a more detailed response to "why" questions than to "what" questions. The possible effects of the proposed change are narrowed down to the single verification condition Exchange_sort#1.*

<- Done

*The user, knowing the potential effects of these changes, is now ready to continue working with SID toward a problem solution.*

Suggest fully defining EXCHANGE_SORT-> Edit

.
.
.

Exec-> Translate File3.Srt

function Exchange_sort(A:int_array):int_array =

```
begin
  entry N ge 1;
  exit (all I:int,
          I in [1..N] ->   Exchange_sort(A)[I]
                         = Value_of_max(Exchange_sort(A),1,I))
       and IsPerm(A, Exchange_sort(A));
  var B:int_array := A;
  var K:int := N;
  keep  K in [1..N];
  loop
    assert (all I:int, I in [k+1..N] -> B[I] = Value_of_max(B,1,I))
      and K in [1..N]  and  IsPerm(A,B);
    if K = 1 then leave end;
    B := Exchange(B,Location_of_max(B,1,K),K);
    k := K-1;
  end;
  result := B;
end;

function Value_of_max(A:int_array; I,J:int):int =
begin
  exit ( assume ( all k:int,
      k in [1..J] and I in [1..N] and J in [1..N]
        -> A[k] le Value_of_max(A,I,J) )
    and ( all I,m:int,
      I in [1..J] and m in [1..J] and I in [1..N] and J in [1..N]
        -> Value_of_max(Exchange(A,I,m),I,J)  =
          Value_of_max(A,I,J) ) );
end;

function Location_of_max(A:int_array; I,J:int):int =
begin
  entry I in [1..N] and J in [1..N] and I le J;
  exit Location_of_max(A,I,J) in [1..J]
    and A[Location_of_max(A,I,J)] = Value_of_max(A,I,J);
  pending
end;

function Exchange(A:int_array; I,J:int):int_array =
begin
  entry I in [1..N] and J in [1..N];
  exit  (all k:[1..N],
           k ne I and k ne J -> Exchange(A,I,J)[k] = A[k] )
```

```
    and IsExchanged(A,Exchange(A,I,J),I,J)
    and IsPerm(A,Exchange(A,I,J));
  pending
end;

function IsExchanged(A,B:int_array; I,J:int):boolean =
begin
  exit (assume IsExchanged(A,B,i,j) iff
              (  I in [1..N]  and  J in [1..N]
              and A[I]=B[J] and A[J]=B[I] ) );
end;

function IsPerm(X,Y:int_array) : boolean =
begin
  exit (assume
          (all Z:int_array, IsPerm(Z,Z))
        and (all Z:int_array,
              IsPerm(X,Z) and IsPerm(Z,Y) -> IsPerm(X,Y)));
end;
```

```
No syntax errors detected.
No semantic errors detected.
```

*Figuring out how to fit this data into the data base involves observing relationships that have disappeared, observing some new ones that are introduced, and using deduction to decide what prior relationships still hold.*

```
Exec-> Print Status All

The current design and verification status is:
   Waiting for VC generation (must check VCs): EXCHANGE_SORT
   Waiting for pending body to be filled in: EXCHANGE, LOCATION_OF_MAX
   For specifications only: ISEXCHANGED, ISPERM, VALUE_OF_MAX
   Constants/Types: INT_ARRAY, N
```

*The system knows that the set of verification conditions for Exchange_sort may now be changed. This is reflected by the phrase "must check VCs" in the status summary. It also indicates that Exchange and Location_of_max are partially defined.*

```
Exec-> Suggest
```

```
Suggest generating VCs for EXCHANGE_SORT-> §
```

*The user accepts the suggestion to continue the verification of Exchange_sort.*

```
Generating new verification conditions for EXCHANGE_SORT ...


-----------------------------------------------
Beginning new path...
Continuing in LOOP ...

Assume (from last assertion)
        (all I#1 : INT,    I#1 in [K+1..N]
                    -> B[I#1] = VALUE_OF_MAX(B, 1, I#1))
    and K in [1..N]
    and ISPERM(A, B)
      .
      .
      .

End of path
-----------------------------------------------

Unaffected VCs: EXCHANGE_SORT#1, EXCHANGE_SORT#2, EXCHANGE_SORT#3
```

*Only new paths and new VCs are displayed, while previous ones that are unaffected by changes are kept intact. Here, additional VCs arise because the "pending" in Exchange_sort was replaced by executable code. Determining that VCs are unaffected involves logical as well as semantic considerations.*

```
Suggest proving VC called EXCHANGE_SORT#4-> Print Status All

The current design and verification status is:
  Waiting for pending body to be filled in: EXCHANGE, LOCATION_OF_MAX
  For specifications only: ISEXCHANGED, ISPERM, VALUE_OF_MAX
  Constants/Types: INT_ARRAY, N

  EXCHANGE_SORT
    Waiting to be proved: EXCHANGE_SORT#4, EXCHANGE_SORT#5,
      EXCHANGE_SORT#6, EXCHANGE_SORT#7
    Proved in theorem prover: EXCHANGE_SORT#1, EXCHANGE_SORT#2,
      EXCHANGE_SORT#3
```

**Suggest proving VC called EXCHANGE_SORT#4-> §**

.
.
.

*The remaining VCs for Exchange_sort are proved. Then, the design and verification of Location_of_max and Exchange is continued, following a scenario similar to the one shown above for Exchange_sort. A bug is detected in Location_of_max and then removed by paralleling the scenario for making any other kind of change. The session is rejoined after completing these tasks.*

**Suggest one of:**
**<esc> TRANSLATE (from file)**
**SAVE (problem on file)**

*The system, knowing that the problem as now represented is solved, presents a menu containing two reasonable suggestions.*

**-> Print Status All**

**The current design and verification status is:**
**Proved: EXCHANGE, EXCHANGE_SORT, LOCATION_OF_MAX**
**For specifications only: ISEXCHANGED, ISPERM, VALUE_OF_MAX**
**Constants/Types: INT_ARRAY, N**

*This summary shows that the design and verification of the sorting program is complete.*

**Suggest one of:**
**<esc> TRANSLATE (from file)**
**SAVE (problem on file)**

**-> Print Unit All**

*Rather than accepting a proposed operation, a display of all program units in the system is requested.*

```
const N : INT = PENDING;

type INT_ARRAY = array ([1..N]) of INT;

function EXCHANGE_SORT(A : INT_ARRAY) : INT_ARRAY =
begin
```

```
  entry N ge 1;
  exit     (all I : INT,    I in [1..N]
                    ->   EXCHANGE_SORT(A)[I]
                      = VALUE_OF_MAX(EXCHANGE_SORT(A), 1, I))
       and ISPERM(A, EXCHANGE_SORT(A));
  var B : INT_ARRAY := A;
  var K : INT := N;
  keep K in [1..N];
  loop
    assert     (all I#1 : INT,    I#1 in [K+1..N]
                          -> B[I#1] = VALUE_OF_MAX(B, 1, I#1))
          and K in [1..N]
          and ISPERM(A, B);
    if K=1 then leave end;
    B := EXCHANGE(B, LOCATION_OF_MAX(B, 1, K), K);
    K := K-1
  end;
  RESULT := B
end;
```

.
.
.

*The final version of all programs and specifications are printed from the system data base in an easy-to-read format and, for this command, in the order in which they were originally entered.*

```
Suggest one of:
  <esc> TRANSLATE (from file)
  SAVE (problem on file)

 -> Print Lemma All


 LEMMA#1 is
   all L : INT,
     all P : INT_ARRAY, L in [1..N] -> VALUE_OF_MAX(P, L, L) = P[L]
```

.
.
.

```
 LEMMA#4 is
   all L, M : INT,
```

all P : INT_ARRAY, L in [1..N] and M in [1..N]
-> ISPERM(P, ALPHA(ALPHA(P, L, P[M]), M, P[L]))

*These are the lemmas used in the verification of the sorting program.*

Suggest one of:
<esc> TRANSLATE (from file)
SAVE (problem on file)

-> Explain

*The explanation facility is useful not only during a design and verification, but also afterwards as a maintenance tool. For example, it may be desirable or necessary to revise an incomplete specification, or to recode an individual program for efficiency. Figuring out the effects of such changes is a formidable task even for this small example, especially after a period of time has elapsed. The following dialog emphasizes its utility on a varied set of questions.*

<- What are the effects of changing the header of IsPerm?

Possible effects of changing the header of ISPERM are:
Incorrect semantics: EXCHANGE, EXCHANGE_SORT, ISPERM, LEMMA#4
Invalid verification: EXCHANGE, EXCHANGE_SORT

<- Why is Exchange affected?

Incorrect semantics because EXCHANGE calls ISPERM. Invalid verification because basis property LEMMA#4, which calls ISPERM, was used in proving EXCHANGE#1.

<- Why is Exchange_sort affected?

Incorrect semantics because EXCHANGE_SORT calls ISPERM. Invalid verification because the external specifications of EXCHANGE, which call ISPERM, were used in proving EXCHANGE_SORT#7. Invalid verification because the external specifications of ISPERM, which recursively call ISPERM, were used in proving EXCHANGE_SORT#1 and EXCHANGE_SORT#7.

<- What are the effects of modifying Lemma#1?

Changing basis property LEMMA#1 does not affect the verification if
    Changed lemma
 -->
  C1: all L : INT,
        all P : INT_ARRAY, L in [1..N] -> VALUE_OF_MAX(P, L, L) = P[L]
*is true. If not, the change invalidates the verification of* EXCHANGE_SORT and LOCATION_OF_MAX.

<- Why?

Because it was used in the proof of EXCHANGE_SORT#3 and LOCATION_OF_MAX#1.

<- What are the effects of changing the exit assertion of IsExchanged?

No effects on ISEXCHANGED. The verification of other programs is not affected if the formula
    Changed exit specification
 -->
  C1:    ISEXCHANGED(A, B, I, J)
      iff I in [1..N] and J in [1..N] and A[I] = B[J] and A[J] = B[I]
*is true. If not, the change invalidates the verification of* EXCHANGE and EXCHANGE_SORT.

<- Why?

No effects on ISEXCHANGED because it is for specifications only and does not have VCs. Invalid verification because properties from ISEXCHANGED were used in proving EXCHANGE#1 and EXCHANGE_SORT#7.

<- What are the effects of changing the entry assertion of Location_of_max?

Changing the entry specification of LOCATION_OF_MAX does not affect LOCATION_OF_MAX if the formula
    Changed entry specification
 -->
  C1: I in [1..N]
  C2: J in [1..N]
  C3: I le J
and the formula

H1: I in [1..N]
H2: J in [1..N]
H3: I le J
-->
Changed entry specification
are true. If not, at least one of LOCATION_OF_MAX's VCs is incorrect. The verification of other programs is not affected if the second formula is true. If not, at least one VC is incorrect for EXCHANGE_SORT.

<- Why?

If the first formula is false, one of LOCATION_OF_MAX's VCs is changed. If the second formula is false, incorrect VCs because the entry specification of LOCATION_OF_MAX is proved in EXCHANGE_SORT and LOCATION_OF_MAX.

<- What does changing its exit assertion affect?

*"It" refers to the program mentioned in the last "what" question.*

Changing the exit specification of LOCATION_OF_MAX does not affect LOCATION_OF_MAX if the formula
H1: LOCATION_OF_MAX(A, I, J) in [I..J]
H2: A[LOCATION_OF_MAX(A, I, J)] = VALUE_OF_MAX(A, I, J)
-->
Changed exit specification
is true. If not, some of LOCATION_OF_MAX's VCs are incorrect. The verification of other programs is not affected if the formula
Changed exit specification
-->
C1: LOCATION_OF_MAX(A, I, J) in [I..J]
C2: A[LOCATION_OF_MAX(A, I, J)] = VALUE_OF_MAX(A, I, J)
is true. If not, the change invalidates the verification of EXCHANGE_SORT.

<- Why?

The effects on LOCATION_OF_MAX are a change in its VCs. Invalid verification because properties from LOCATION_OF_MAX were used in proving EXCHANGE_SORT#5 and EXCHANGE_SORT#7.

**<- Done**

**Suggest one of:**
**<esc> TRANSLATE (from file)**
**SAVE (problem on file)**

**-> Save**

**Enter file name-> Sort.Dmp**

*The session is finished and the current problem state is saved on file Sort.Dmp in a format suitable for restoration.*

# CHAPTER 2

# RELATED RESEARCH

This chapter presents a general review of six previous program verification systems, then compares them to SID. Although it is difficult to characterize differences between programs, an accurate description of their underlying philosophy and their capabilities will be attempted. These impressions have not been reviewed by the author(s) of the systems discussed herein.

## 2.1 Program Verification Systems

King [69,71] describes the initial program verification system. It is essentially written in assembly language, being the only non-Lisp-based system among the six. This system is fully automatic, allowing no user interaction once a program and its specifications are supplied. King [69, p. 130, p. 166] points out that the inability to interact with the user is a main source of certain system limitations. King's system performs extremely well on the small class of programs to which it applies, including certain algebraic computations, such as multiplication (division) by successive additions (subtractions) and prime determination, and array manipulations like zeroing an array and a simple exchange sort.

Deutsch's system [73] has successfully verified the examples in King's thesis (except the system of linear inequalities in Example 10) , plus various other programs such as FIND [Hoare 71]. This included King's Example 9, which is a sorting program King was unable to verify due in part to the absence of a few critical assertions. This system contains a program editor, allows for some user interaction, and has in part a resolution-based deductive system. Deutsch states that his system's "major departure from King's system arises in the theorem generating process" [Deutsch 73, p. III-3]. King utilizes a backward approach to VC generation that often results in lengthy, unmanageable VCs. Deutsch's system generates VCs in a forward manner, very similar to methods originally described by Good [70]. The main idea is to perform certain deductions in parallel with the generation to produce greatly simplified VCs.

The SRI system [Elspas, et al. 73, Waldinger and Levitt 74] differs from the other systems particularly with respect to their deductive system, written in QA4 [Rulifson, et al. 72]. Since QA4 incorporates a considerable variety of sophisticated features (e.g., pattern matching

against tuples, bags, and sets, backtracking, goal-driven deduction, and demons), their powerful theorem prover, which uses depth-first search, required only about ten pages of code [Elspas, et al. 73, p. 43]. The interactive philosophy of this system is that the user grants permission to allow the verifier to proceed; it has been used to verify a varied and interesting set of examples. This set includes various algebraic computations, array sorting and rearrangement programs, and parts of a pattern matcher and unification algorithm.

The Stanford University system [von Henke and Luckham 75, Luckham and Suzuki 75, Suzuki 75] has verified many important benchmark programs, including several sorting programs (proving termination and worst case timebounds in addition to the standard sorting properties), programs that operate on pointers [Luckham and Suzuki 76], and fairness of a simple queuing system [Karp and Luckham 76]. The VC generator is that of Igarashi, et al. [73]. The system's powerful proving facility includes an arithmetic, conditional, and unconditional simplifier, a goal-subgoaler, plus other valuable features used in conjunction with a resolution-based theorem prover. User interaction generally occurs only when the user desires to add certain rules or strategies applicable to the domain of interest; they are subsequently invoked automatically.

The USC Information Sciences Institute/University of Texas at Austin system [Good, et al. 75] is highly interactive and has verified several sorting and searching programs, small routines from the verifier and from a compiler (all converted to a subset of Pascal), as well as examples from King's thesis. The design strategy was to "provide automatic capability for the proof process where possible, and to rely on interaction for manual intervention otherwise" [Good, et al. 75, p. 483]. The authors believe the main unique features of their system to be "its good facilities for user interaction, the modular system design which uses several previously existing components, the particular natural deduction prover that is used, and the theorem prover's method of incremental bounding of values of variables, which, among other things, facilitates automatic proof by cases" [Good, et al. 75, p. 483]. The VC generator of this system is essentially the one of Igarishi, et al. [73], originally developed at Stanford; and its theorem prover is a variation of the prover discussed in Bledsoe and Bruell [73], originally developed at UT Austin.

Each of these five systems can, at a first approximation, be seen to consist of four main components: a parser, VC generator, simplifier, and theorem prover. The approach to each of these phases of verification differs somewhat from system to system, as does the amount of user interaction. All are based on the conventional inductive assertion method, and all (excepting King) provide facilities for handling functional abstraction.

Another notable verification system, not based on inductive assertions, is the Boyer-Moore Lisp proving system [Moore 73, 75, Boyer and Moore 75, 77]. Based on structural induction, it uses Lisp for stating both programs and specifications. Its proof system includes heuristics for theorem generalization, rewrite rules, equality substitution rules, and a structural

induction heuristic. Its basic strategy is to EVAL an expression via the Lisp interpreter and, if it is not T, to apply parts of the proof system, then reiterate the process; when the expression EVALs to T, the proof is complete. It has automatically proved many interesting Lisp properties and a simple optimizing compiler for expressions. A typical Lisp property proved is that REVERSE is its own inverse, i.e., that (EQUAL (REVERSE (REVERSE A)) A) is TRUE.

Other verification systems with published descriptions include: Carter, et al. [77] at IBM T. J. Watson Research Center, Crocker[77] and Musser [77] at USC Information Sciences Institute, Hookway [76, Ernst and Hookway 75] at Case, Marmier [75] at ETH in Zurich, Schorre [76] at SDC, and Topor [75] at Edinburgh. Also of interest is Good [70], which implemented certain useful bookkeeping facilities as part of a verification system that accepted only hand proofs.

## 2.2 Comparison with Previous Systems

Four broad areas for comparing SID with other systems will be used. Of course, no small set of categories can completely characterize the differences between programs, but they can provide some viewpoints to help see past superficial differences.

### Toward a New Paradigm

Paradigms are a good way of looking at a class of important differences between SID and earlier systems. In the past, systems for assisting in producing verified programs have been, on the whole, non-incremental and focused on verification alone. Looking at the task in this way imposes several sometimes impractical constraints on the way things can be done. To lift these constraints, SID appeals to a new paradigm -- one that adopts an incremental, integrated view of program design and verification.

It is easy to illustrate the inadequacy of the non-incremental view. Consider a programming problem whose solution consists of several programs and their associated specifications. According to this view, all programs are, in essence, completely designed and then verified separately afterwards. The overall verification is complete when all individual programs are verified. Or is it? What happens if modifications, like changing a specification or the type of an identifier, were made along the way? To guarantee consistency, the entire verification must be re-started from the beginning, or the user must assume the burden of determining the impact of changes, or a combination of both. Most often, the solution is the objectionable activity of redoing everything, since placing the burden on the user is viable only in the simplest cases.

What additional kinds of knowledge does SID have to enable it to lift constraints imposed by this non-incremental view? Previous systems are knowledgeable about how to do individual verification tasks, such as how to generate VCs. But they are not knowledgeable enough about how a verification fits together (e.g., what specifications are in what VCs, what properties are assumed where, what functions are called by what verification-related information), how a design fits together (e.g., dependencies like who calls whom, or where is X assumed to be of type Y), and how design and verification interrelate. SID brings together these kinds of issues.

### User Interface

Attention to user interface issues is important in any interactive system intended for actual use. The degree of its importance, of course, varies with the domain; it is accentuated here due to the complexity and incremental nature of the task. User/system interaction may occur often, and for a wide variety of reasons.

How does SID respond to such issues? Unlike other systems, its command structure is based on a dual-mode philosophy that, according to user preference, is active in helping to figure out what to do next, or is passive and simply responds to each input, then waits for the next. Help is provided in the form of system-generated suggestions, which are always based on the current state of development. Alternatively, the user is given the option of directing and querying the system via a comprehensive set of available commands.

The overall effectiveness of SID is also significantly enhanced by its ability to engage in an interactive English dialog about the potential effects of changes, as shown in the example session of Section 1.2.

### Programming Languages and Proof Methods

Programming languages supported by earlier systems are subsets of languages such as Pascal, Algol, or Lisp.

SID, on the other hand, supports the language Gypsy which has three important features that these other languages do not have. The first is its consistent integration of program and specification statements. This allows SID to parse and type check programs, specifications, and properties assumed in proofs. Second, it consolidates compile-time and run-time specifications, enabling the system to take full advantage of the interrelationship between verification and run-time validation by assuming the specifications to be checked at run-time in the verification. Third, it contains constructs for error handling and for stating and specifying data abstractions and concurrent processes, extending the kinds of programs that can

be considered. Gypsy also eliminates a few constructs, such as go-to statements and labels, that are handled in some of the other systems.

Concerning proof methods, five of the systems reviewed are based on inductive assertions, and Boyer-Moore uses structural induction. Other methods can be found in a few systems not reviewed here, e.g., computational induction [Milner 72] and continuation induction [Topor 75]. SID employs inductive assertions, plus the methods described in Good [77] for proving data abstractions and concurrent processes.

**Verified Programs**

Programs verified in previous systems are sequential and characteristically small in size, yet often logically complex. They include certain array sorting and rearrangement programs, algebraic computation programs such as square root, quotient, factorial, and exponentiation, and tree searching, pattern matching, and unification programs.

An advantage of studying these previously-used examples is that generally they are simple enough to be easily grasped. This is good for expository purposes, but there is a danger in considering only these since they do not provide a very good metric on what else systems can do. Therefore, the focus here is on moving into a very different kind of area in which there has been little if any prior work -- communications processing systems. The intent is to apply SID to large, realistic programs, thereby establishing a better base from which to judge its practical utility and philosophy.

Currently, the message switching network in Appendix A is the largest and most complex program completely designed and verified using SID.

# CHAPTER 3

# THE SYSTEM

## 3.1 System Overview

Figure 3-1 illustrates the high-level system design; this is also how the user views the overall system. SID consists of four parts: a designer/verifier's assistant (or simply assistant), a translator, a VC generator, and a proof system. The human designer/verifier communicates with the assistant, which invokes the translator for parsing or type checking, the VC generator for generating VCs, or the proof system for proving a formula. The assistant is the new component in this scenario; some form of the other components have traditionally been included in earlier verification systems (cf. Chapter 2). The idea behind the assistant is that the user, instead of talking to a passive system that responds to each input and then waits for the next, talks to an active, helpful intermediary. The assistant embodies an incremental, integrated view of design and verification and ties together the other components to convey this view to the user.

The assistant is organized as shown in Fig. 3-2. Its user interface is discussed in Section 3-2, which also gives an overview of SID's explanation facility as a prelude to a discussion of the natural language part of the interface. Section 3-3 describes all of SID's incremental capabilities. The majority are provided solely by the assistant, specifically the design and verification manager (abbreviated as DVM), using the methods of Chapter 4. The others require interactions which are discussed here between the DVM and other components. The assistant's data base, whose organization is presented in Section 4.2, is often referred to as simply "the data base". Section 3-4 focuses on SID's design, discussing its important overall characteristics and the functional capabilities of the traditional system components. The impact of an integrated, incremental view of design and verification on SID's design is emphasized. Some implementation details (e.g., SID's memory requirements) are presented in Section 3-5. The example session in Section 1.2 is referred to throughout this chapter.

## 3.2 User Interface: Commands, Suggestions, and English Queries

SID interacts with the user by accepting and executing commands, by proposing

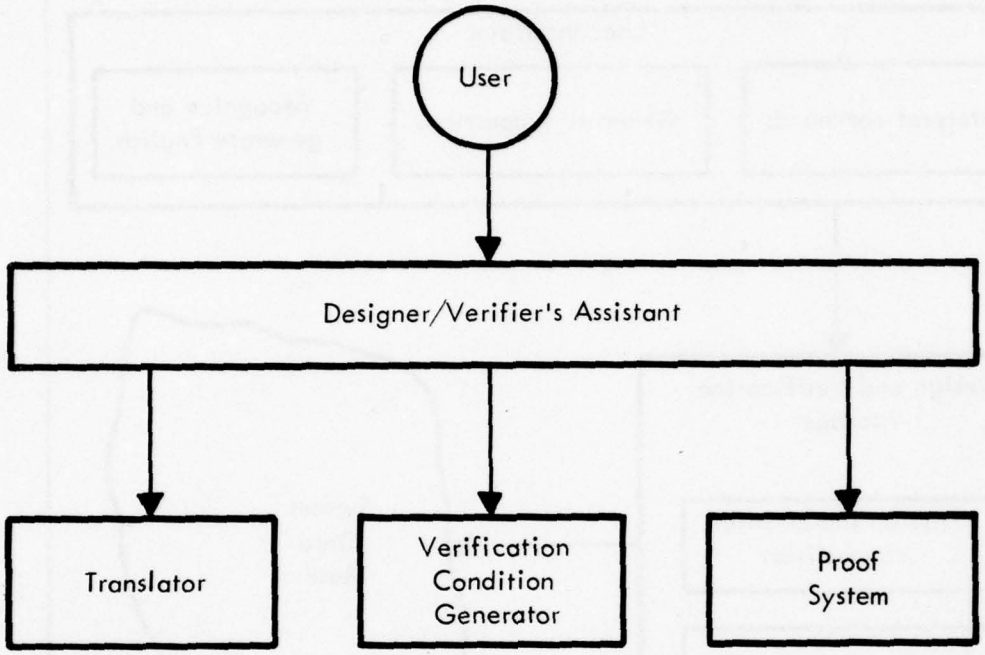



Figure 3-1. High-level structure of SID

actions, and by conversing about the effects of changes. Each of these aspects of its user interface is discussed in this section.

**Command Language**

SID's command language includes several kinds of control, help, print, and query commands. Each command begins with an initial keyword that identifies its main function, as summarized in Fig. 3-3. For variations in command execution, some commands allow subcommands, which are further keywords that select options or arguments that tell what to act on. The print command, for example, is used to display user-defined data (e.g., individual or collections of programs, specifications, assumed properties, or VCs) and system-generated data (e.g., status summaries).

Full and abbreviated input styles are allowed. The user can type any keyword in its entirety or type an unambiguous abbreviation of it. Arguments can also be abbreviated and sometimes even omitted.

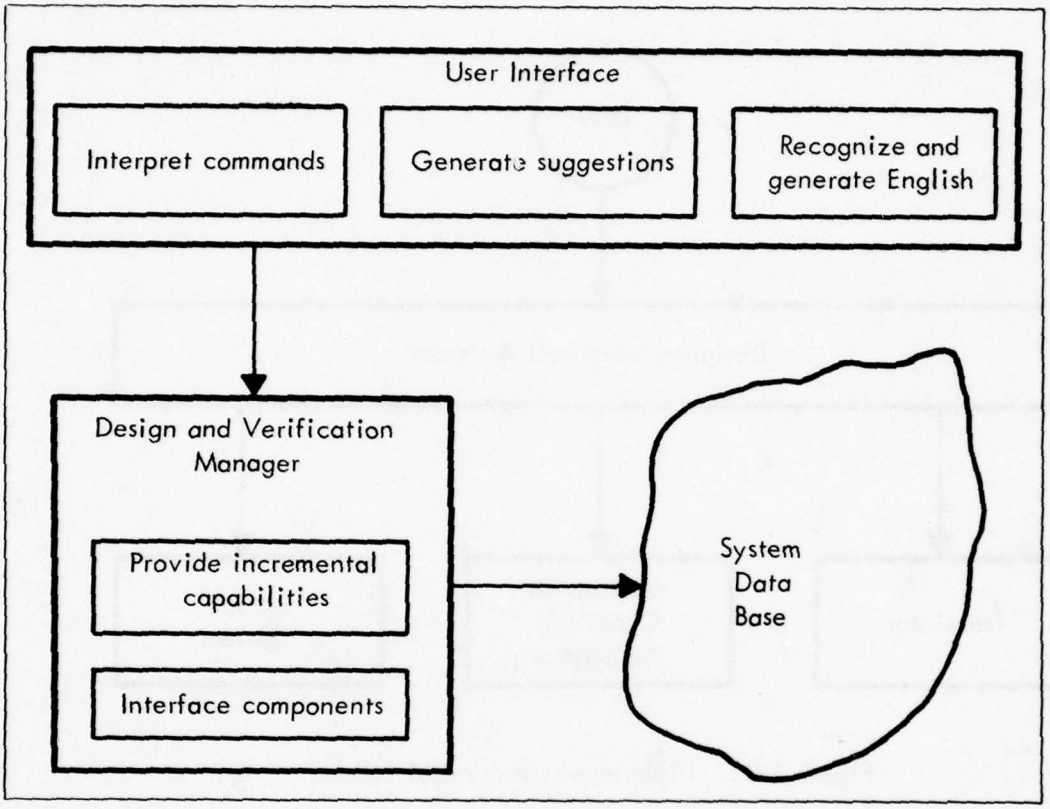



Figure 3-2. Structure of Designer/Verifier's Assistant

Default assignments are automatically supplied in some contexts by SID if the user declines to specify a particular argument himself. The heuristic employed is to default to the last legal program name typed by the user or by the system in a suggestion, or to the first verifiable program on the last translated file, whichever is most recent. This heuristic encourages a top-down approach to design and verification. In the sorting program development in Section 1.2, the default is initially Exchange_sort, then Location_of_max, then Exchange. The impact of this heuristic is even more evident in the more lengthy network development in Appendix A.

**Making Suggestions to the User**

The suggestion mechanism is valuable for experienced as well as novice users of SID. It can guide the user through the design and verification of an entire program, whether it

| Command | Function |
|---|---|
| ADD | Parse and type check new lemma |
| DELETE | Delete lemma |
| EDIT | Invoke text editor * |
| EXPLAIN | Enter explanation facility |
| HELP | Explain command(s) |
| PRINT | Display information |
| PROVE | Attempt proof of formula |
| RESTORE | Restore a previously-saved problem state |
| SAVE | Save the current problem state |
| SUGGEST | Propose an action * |
| TRANSLATE | Parse and type check data on file |
| VCS | Generate VCs |
| ↑E (control E) | Delete current word * |
| ? | List currently allowed options * |
| ?? | List full syntax for remaining allowable options |

* indicates no subcommands available

Figure 3-3. Summary of commands

consists of just a few programs or of a sizeable, perhaps intellectually unmanageable, number of programs. Three main characteristics of this mechanism are its ability to:

- Always have available a reasonable suggestion for the next step in the design and verification.
- Base suggestions on the current state of development.
- Make suggestions easy to obtain, while not impeding the invocation of other desired operations.

Some design issues relating to the first two characteristics are discussed before looking at the actual implementation; the last is accomplished by the straightforward convention for transferring between command and suggest mode illustrated in Section 1.2. A similar mechanism is described in Yonke [76].

The key issue involved in generating suggestions is that of *priorities* -- what suggestions should be given when. Representative issues confronting the suggestion mechanism are: Should operations on partially-defined programs be suggested before or after operations

on fully-defined ones? Is their manipulation interspersed rather than sequential? Should completing the proof of partially-proved VCs be suggested before or after trying completely unattempted proofs is suggested? What about programs verified in varying degrees? What impact do incremental changes have on suggestion generation?

This priority issue is addressed here by basing the order of suggestions on a pre-imposed priority class structure. The scheme for keeping track of suggestions is naturally a priority queue. Each suggestion is assigned a fixed priority number when it is created, then placed in the queue ahead of lower priority suggestions and behind suggestions of equal or higher priority. The next suggestion is always at the front of the queue. If a suggestion is executed or no longer appropriate, it is removed from the queue.

To illustrate the suggestion mechanism in action, the queue manipulations that took place during the development in Section 1.2 will be traced, accompanying each manipulation by the suggestion SID made (or would have made had it been in suggest mode). The priority queue is represented as an ordered list of sublists, where each sublist is a triple of the form

```
(priority    operation    object)
```

The queue is empty when it contains no such entries.

When the session begins, the suggestion queue is empty and SID types

```
Suggest one of:
  <esc>TRANSLATE (from file)
  RESTORE (problem from file)
```

The initial version of Exchange_sort plus data definitions N and Int_array are entered into the system. The updated queue is

```
( (5  CorrectSemantics  Exchange_sort) )
```

N and Int_array do not have queue entries because they are not to be verified. The suggestion typed is

```
Suggest correcting the semantics of Exchange_sort
```

indicating the need to correct the type errors in Exchange_sort. After this is done, the queue is updated to

```
( (5  GenerateVCs  Exchange_sort) )
```

and SID types

**Suggest generating VCs for Exchange_sort**

Execution of this operation results in three VCs and the discovery of a pending path.

```
( (5    Prove         Exchange_sort#1)          (3.1)
  (5    Prove         Exchange_sort#2)
  (5    Prove         Exchange_sort#3)
  (4    FullyDefine   Exchange_sort ) )
```

Priority assignments encourage top-down development and thus complement the scheme for making default assignments in commands. Here, proving existing VCs has a higher priority than continuing the design. The following suggestions are made, accepted, and successfully carried out in sequence.

**Suggest proving VC called Exchange_sort#1**
**Suggest proving VC called Exchange_sort#2**
**Suggest proving VC called Exchange_sort#3**

The remaining queue entry is

```
( (4    FullyDefine    Exchange_sort) )          (3.2)
```

So SID suggests continuing the design by typing

**Suggest fully defining Exchange_sort**

The user completes the implementation of Exchange_sort and also makes several other changes to the overall design. The only ones affecting the suggestion queue are the introduction of functions Location_of_max and Exchange.

```
( (5    GenerateNewVCs    Exchange_sort )
  (4    FullyDefine       Location_of_max)
  (4    FullyDefine       Exchange      ) )
```

The next suggestion is

**Suggest generating new VCs for Exchange_sort**

After four new VCs are generated, the queue entries are

```
( (5    Prove        Exchange_sort#4)
  (5    Prove        Exchange_sort#5)
  (5    Prove        Exchange_sort#6)
  (5    Prove        Exchange_sort#7)
  (4    FullyDefine  Location_of_max)
  (4    FullyDefine  Exchange      ) )
```

After suggestions to prove these VCs are given and carried out, suggestions parallel those beginning with queue configuration (3.2). Location_of_max, followed by Exchange, is implemented and verified. Afterwards, the suggestion queue is empty and SID types

```
Suggest one of:
  <esc> TRANSLATE (from file)
  SAVE (problem on file)
```

When the queue is empty, suggestions are based on whether or not the system data base is empty. The data base is empty only at the beginning of a problem, whereas the suggestion queue is empty at the beginning and at the end.

A slightly complicating factor -- which is a by-product of SID's dual-mode philosophy -- not encountered in the above scenario is the problem of updating the suggestion queue when the user issues a command whose execution affects an entry (or entries) already in the queue. As an illustration of how the queue is manipulated, suppose that the queue is configured as in (3.1) and that the user, instead of accepting the suggestion to prove Exchange_sort#1, modifies the implementation of Exchange_sort. Then, all entries relating to Exchange_sort are removed and a new suggestion such as

```
( (5    GenerateNewVCs    Exchange_sort) )
```

is added. When VCs are generated for Exchange_sort, entries for those VCs that remain valid are reentered.

### Conversing about the effects of changes

SID's explanation facility answers questions about what the effects of changes are and about why a change has the determined effects. When the user types a question, SID goes through three steps:

- ***Understanding the question.*** A simple pattern matching scheme which looks for keywords works well for SID's limited domain of discourse. Context is used only in resolving pronoun references. These patterns of

keywords are translated into calls to functions responsible for answering the question. If a question is not understood, the user is informed of the allowable options for the missing or incorrect sentence fragments.

- ***Getting the answer.*** This involves two interleaved processes -- figuring out the answer and formatting it. Functions, called from Step 1, interface with the methods of Chapter 4 to determine the effects of changes and generate the appropriate English output by fitting the results of the methods into language templates. These templates are assembled dynamically from smaller, standardized templates that are designed to fit very specific situations. Answers to "why" questions are more detailed in some respects than answers to "what" questions.

- ***Reporting the answer.*** A set of routines for formatted printing of templates is used for typing the English answer on the terminal.

Figure 3-4 illustrates these steps. This section focuses on the natural language interface portion of the explanation facility.

The workings of this English interface are illustrated using an excerpt from the final conversation in Section 1.2 (shown in Fig. 3-5). Figure 3-6 explains the role of keywords used in forming the function calls needed to get the answers. Although not used in function call formation, verbs are also recognized. This is done to increase the likelihood of the answer making sense. The dictionary of keywords contains verbs meaning "alter" (e.g., alter, change, modify, and replace) in several tenses. Figure 3-7 shows how final answer templates are built by filling in intermediate templates, which represent part of the answer, with actual problem-specific data. The methods of Chapter 4 are used at each step to guide the choice of templates and to supply actual data for filling them in. The answer template is then typed on the terminal as English text by routines designed especially for this purpose. Notice in Fig. 3-5 that the answer to the "what" question identifies the program Exchange_sort as potentially being affected by the change, while the answer to the "why" question isolates the potential effects to VCs Exchange_sort•5 and Exchange_sort•7.

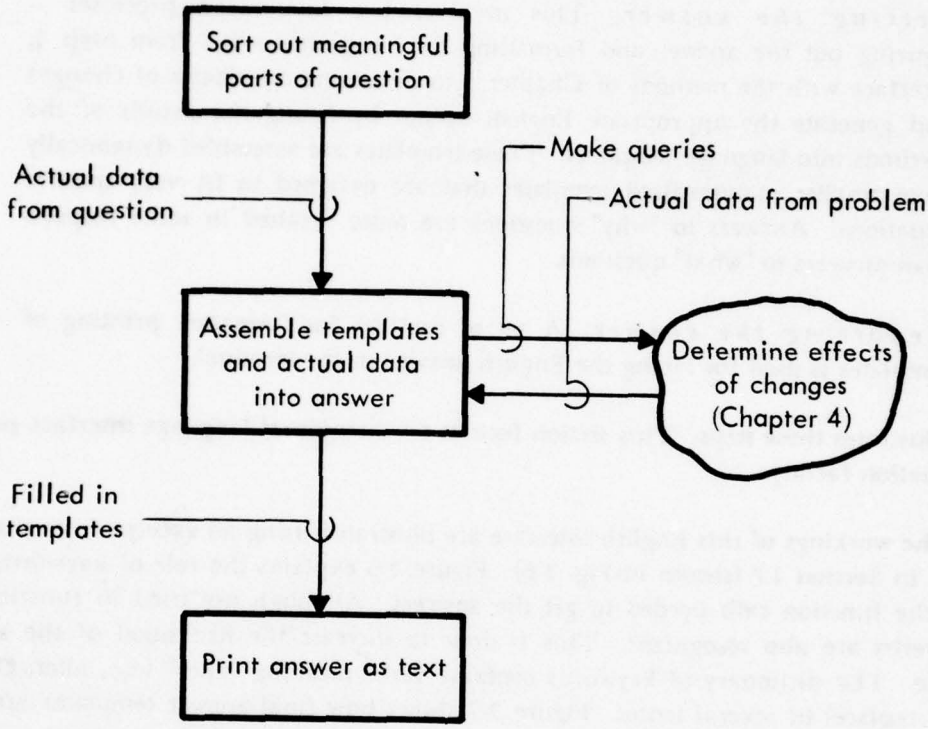



Figure 3-4. Structure of the explanation facility

<- What are the effects of changing the entry assertion of Location_of_max?

.
.
.

<- What does changing its exit assertion affect?

Changing the exit specification of LOCATION_OF_MAX does not affect
LOCATION_OF_MAX if the formula
  H1: LOCATION_OF_MAX(A, I, J) in [I..J]
  H2: A[LOCATION_OF_MAX(A, I, J)] = VALUE_OF_MAX(A, I, J)
 -->
    Changed exit specification
is true. If not, some of LOCATION_OF_MAX's VCs are incorrect. The verification
of other programs is not affected if the formula
    Changed exit specification
 -->
  C1: LOCATION_OF_MAX(A, I, J) in [I..J]
  C2: A[LOCATION_OF_MAX(A, I, J)] = VALUE_OF_MAX(A, I, J)
is true. If not, the change invalidates the verification of EXCHANGE_SORT.

<- Why?

The effects on LOCATION_OF_MAX are a change in its VCs. Invalid verification
because properties from LOCATION_OF_MAX were used in proving EXCHANGE_SORT.5
and EXCHANGE_SORT.7.

Figure 3-5. Sample questions and answers

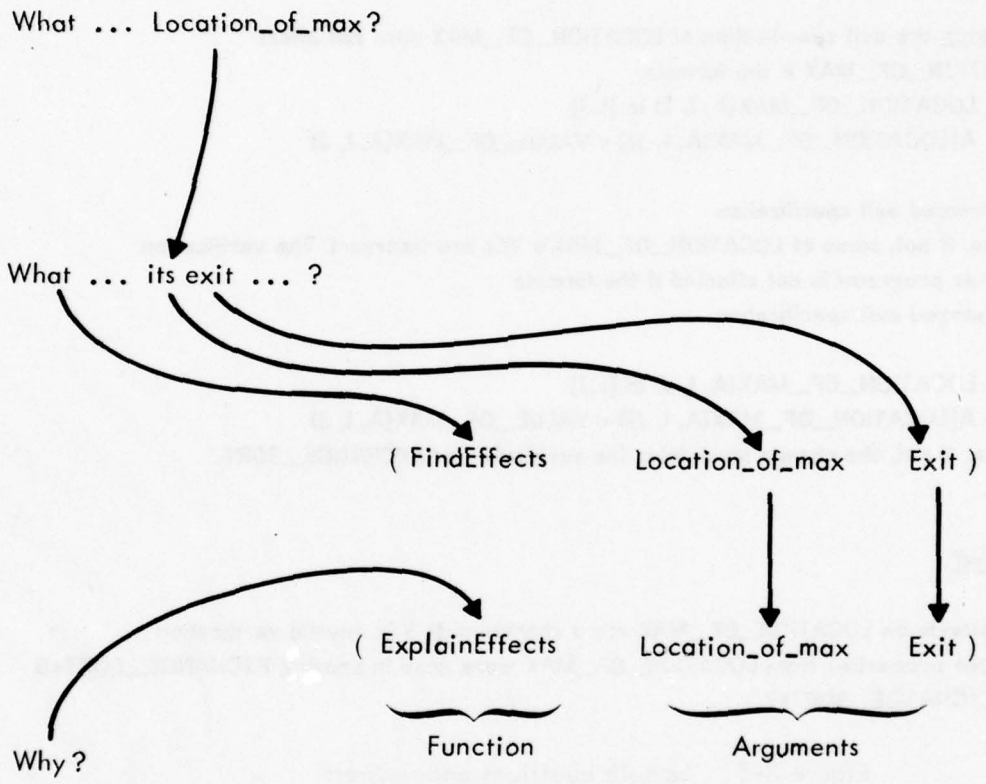



Figure 3-6. Translating questions of Fig. 3-5 into function calls

**The verification of other programs is not affected if the formula**

**Changed exit specification**

**→**

**C1: LOCATION_OF_MAX(A, I, J) in [I..J]**

**C2: A[LOCATION_OF_MAX(A, I, J)] = VALUE_OF_MAX(A, I, J)**

**is true. If not the change invalidates the verification of EXCHANGE_SORT.**

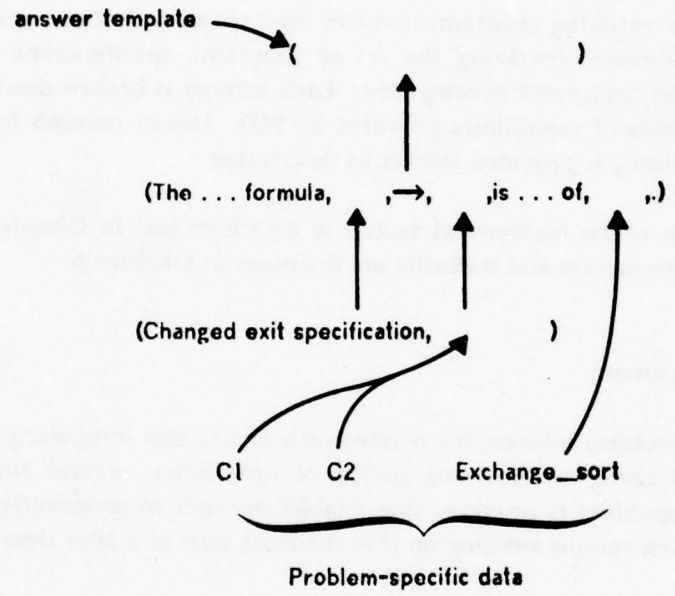



Figure 3-7. Assembling part of answer to "what" question of Fig. 3-6

## 3.3 Incremental Capabilities

One important point needs to be emphasized before going on. SID's incremental capabilities were designed to *collectively* convey an incremental view. They are highly integrated and complementary, as seen in the session in Section 1.2. Thus, the overall scenario should be kept in mind when reading the individual descriptions below.

Designing and verifying programs involves four major activities -- overall problem development, design refinement (evolving the set of programs, specifications, and assumed properties), VC generation, and proof development. Each activity is broken down according to the user's view of the kinds of capabilities provided by SID. Unless covered in Chapter 4, a discussion of how a capability is provided follows its description.

The discussion of the incremental aspects of SID here and in Chapter 4 focuses on what was done. Some alternatives and tradeoffs are discussed in Chapter 5.

### Overall Problem Development

Obtaining a problem solution, for reasons such as size and complexity, is typically an evolutionary process requiring an extended period of time (often several sittings with the system). Therefore, a capability is provided that enables the user to temporarily interrupt the problem development, then resume working on it in the same state at a later time.

If the user types

```
Exec-> Save Sort.Dmp
```

the DVM saves the current problem state on the file Sort.Dmp. This requires collecting all data that defines the current problem state and then writing it on the file Sort.Dmp in a format suitable for restoration. Typing

```
Exec-> Restore Sort.Dmp
```

causes the save file Sort.Dmp to be read back into the system, replacing the current state with the one recorded on Sort.Dmp.

What information needs to be collected? How can it be collected? Are there any formatting problems? These are some of the questions that often create problems when providing this kind of capability. Due to the overall system design -- in particular, the centralization of information in the assistant's data base -- such questions pose no problems here. The DVM simply writes out this single data base to temporarily discontinue a problem, then reads it back in to resume a problem.

### Design Refinement

Typically, a problem consists of a large, highly-interrelated collection of information, including programs, specifications, and properties assumed in proofs. The focus here is on how SID aids in evolving the final design of this interwoven problem structure.

***Changing the design.*** The user can add new or change already-defined programs, specifications, or properties assumed in proofs, letting SID assume the burden of determining the design-related effects. Changes are made for several reasons -- to correct an error in a program or specification, to augment or reformulate a program or its specifications, to add a new program or assumed property, and so on. SID, by figuring out the impact of changes, allows the user to concentrate on the essentials of determining the appropriate change.

The DVM analyzes changes to the design with the intent of avoiding complete retranslation (and reverification too). Of course, in some cases it cannot be avoided. Changing the number or types of arguments of a function, for example, requires that all calls on the function be checked. Therefore, those programs, specifications, and assumed properties that do not call the changed function need not be (and are not) rechecked. Sections 4.3-4.5 give the relevant algorithms.

***Partially stating programs.*** Just as an incomplete overall design is gradually filled in and refined, programs within the overall design structure are developed incrementally. The user can define parts of a program and leave other parts to be defined later. This aids in doing top-down design and verification. For example, partially defining Exchange as

```
function Exchange(A:int_array; I,J:int):int_array =
begin
  entry I in [1..N] and J in [1..N];
  exit  (all k:[1..N],
           k ne I and k ne J -> Exchange(A,I,J)[k] = A[k] )
    and IsExchanged(A,Exchange(A,I,J),I,J)
    and IsPerm(A,Exchange(A,I,J));
  pending
end;
```

allows its header to be used for type checking Exchange_sort and its specifications to be used in the proof of Exchange_sort. Later, the pending body is replaced by executable code

```
function Exchange(A:int_array; I,J:int):int_array =
begin
  entry I in [1..N] and J in [1..N];
  exit  (all k:[1..N],
           k ne I and k ne J -> Exchange(A,I,J)[k] = A[k] )
    and IsExchanged(A,Exchange(A,I,J),I,J)
    and IsPerm(A,Exchange(A,I,J));
  result := A;
  result[I] := A[J];
  result[J] := A[I];
end;
```

and Exchange is verified.

There are three main uses of pending handled by SID.

*Body of routine and type definitions.* The body of a routine (i.e., a function, procedure, process, or program) or type definition may be left pending in several ways. Example uses in a routine are:

```
function ... = pending;
function ... = begin entry ...; pending end;
function ... = begin var...; pending end;
function ... = begin pending end;
```

Notice that all headers (i.e., the entities to the left of the "=" symbol) are complete so that type checking of callers can be done. The second construction serves the additional purpose of allowing properties to be stated about the pending function that can be used in proofs of calling programs.

Some uses of pending in type definitions are:

```
type x = pending;
type x = begin axiom...; pending end;
type x = begin pending end;
```

The second construction allows the axiom specification to be used for verification purposes, just as routine specifications were made available above.

*Statement list in routine bodies.* A pending statement list in a routine body indicates that a particular section of the body has not been written. Typical uses include:

```
if ... then pending end;
if ... then ... else pending end;
loop pending end;
case ... is ...:pending; is ... :pending; else ... end;
```

Handling a program with a pending body and handling a program with a pending statement list differs concerning the generation of VCs. Maybe some VCs can be generated for programs containing pending statement lists, whereas those with pending bodies have no VCs.

*Constant and variable initializations.* A pending constant or variable initialization value indicates that the programmer has not as yet decided the initial value that should be assigned to the particular entity. A pending constant definition is of the form:

```
const x: typ = pending;
```

This construction is allowed for both global and local constant declarations. A pending variable initialization is of the form

```
var x: typ = pending;
```

These uses of pending allow as yet undetermined constants to be "specified". In VC generation and in the proof process, these pending values are treated as unique symbolic constants. Thus, proofs succeed without knowledge about the value eventually assigned. Notice that pending constants and variables must have declared types for type checking purposes.

## VC Generation

To convey a unified, consistent view of the task to the user, VC generation activities must directly reflect steps taken in the design process. This is done by generating (when possible) some VCs for partially-stated programs and by generating only new VCs.

***Generating VCs for parts of programs.*** When a program is only partially stated, VCs are generated for only those paths that are completely defined. For example, Exchange_sort is defined initially as

```
function Exchange_sort(A:int_array):int_array =
begin
  entry N ge 1;
  exit (all I:int,
          I in [1..N] ->   Exchange_sort(A)[I]
                         = Value_of_max(Exchange_sort(A),1,I))
       and IsPerm(A, Exchange_sort(A));
  var B:int_array := A;
  var K:int := N;
  keep  K in [1..N];
  loop
    assert (all I:int, I in [k+1..N] -> B[I] = Value_of_max(B,1,I))
      and K in [1..N]  and  IsPerm(A,B);
    if K = 1 then leave else pending end;
  end;
  result := B;
end;
```

Even though this definition is incomplete (the else branch of the if statement is pending), some VCs can be generated. As shown below, VCs are generated for the path from the entry of Exchange_sort to its loop assertion and for the path from its loop assertion to its exit, whereas that path through the loop must be handled later when the else branch is fully defined.

```
------------------------------------------------
Beginning new path...
B := A
K := N

Assume (unit entry condition)
    N ge 1
       .
       .
       .
ASSERT     (all I#1 : INT,    I#1 in [K+1..N]
                      -> B[I#1] = VALUE_OF_MAX(B, 1, I#1))
      and K in [1..N]
      and ISPERM(A, B)
- - - - - - - - - - - - - - - - - - - - - - - -
Must verify ASSERT condition
Verification condition EXCHANGE_SORT#1
 H1: 1 le N
 -->
 C1: ISPERM(A, A)
```

```
- - - - - - - - - - - - - - - - - - - - - - -
End of path
-------------------------------------------------



-------------------------------------------------
Beginning new path...
Continuing in LOOP ...

Assume (from last assertion)
       (all I#1 : INT,    I#1 in [K+1..N]
                    -> B[I#1] = VALUE_OF_MAX(B, 1, I#1))
    and K in [1..N]
    and ISPERM(A, B)


Assume (KEEP assertion)
    K in [1..N]


Assume (IF test failed)
    (not K=1)

Entering PENDING statement...cannot continue in this path

End of path
-------------------------------------------------



-------------------------------------------------
Beginning new path...
Continuing in LOOP ...

Assume (from last assertion)
       (all I#1 : INT,    I#1 in [K+1..N]
                    -> B[I#1] = VALUE_OF_MAX(B, 1, I#1))
    and K in [1..N]
    and ISPERM(A, B)
       .
       .
       .
```

```
Leaving unit EXCHANGE_SORT
ASSERT     (all I : INT,    I in [1..N]
                    -> RESULT[I] = VALUE_OF_MAX(RESULT, 1, I))
      and ISPERM(A, RESULT)
- - - - - - - - - - - - - - - - - - - - - - - -
Must verify (unit exit) condition
Verification condition EXCHANGE_SORT#3
  H1: all I#1 : INT, I#1 in [K+1..N] -> VALUE_OF_MAX(B, 1, I#1) = B[I#1]
  H2: K=1
  H3: ISPERM(A, B)
  H4: K in [1..N]
 -->
  C1: all I : INT, I in [1..N] -> VALUE_OF_MAX(B, 1, I) = B[I]
- - - - - - - - - - - - - - - - - - - - - - - -
End of path
-------------------------------------------------
```

The fact that a VC cannot be generated for the else branch (on the second path) is detected by the VC generator.

The previous discussion about the uses of pending mentioned that pending values are treated as unique symbolic constants. This impacts the process of generating VCs. The VC generator, upon recognizing a pending as an initialization value, assigns the associated identifier a unique name (which turns out to be the identifier itself) for which no initial value is substituted. Thus, such identifiers are treated as representing some unknown, yet fixed, value. The VC generator recognizes a pending in local initializations by its ordinary path transversal algorithm. Global constants that have pending values are detected via the DVM, which is aware of all pending entities. The DVM ensures that global names are handled properly not only during VC generation but also during proofs.

***Generate only new VCs.*** Changes to the design often affect the verification. For example, changing a statement in a program can change the set of VCs that need to be proved for that program. Or changing the number of arguments in the parameter list of a function invalidates some (but not necessarily all) VCs of its callers. SID, from the user's viewpoint, responds to changes by generating only new VCs and by not regenerating VCs unaffected by the change.

This capability is illustrated by replacing the pending in Exchange_sort with executable code and then generating its VCs. The new version of Exchange_sort is

```
function Exchange_sort(A:int_array):int_array =
```

```
begin
  entry N ge 1;
  exit (all I:int,
          I in [1..N] ->   Exchange_sort(A)[I]
                         = Value_of_max(Exchange_sort(A),1,I))
       and IsPerm(A, Exchange_sort(A));
  var B:int_array := A;
  var K:int := N;
  keep  K in [1..N];
  loop
    assert (all I:int, I in [k+1..N] -> B[I] = Value_of_max(B,1,I))
      and K in [1..N]  and  IsPerm(A,B);
    if K = 1 then leave end;
    B := Exchange(B,Location_of_max(B,1,K),K);
    k := K-1;
  end;
  result := B;
end;
```

SID traces the one changed path through the loop and generates new VCs.

```
------------------------------------------------
Beginning new path...
Continuing in LOOP ...

Assume (from last assertion)
       (all I#1 : INT,    I#1 in [K+1..N]
                    -> B[I#1] = VALUE_OF_MAX(B, 1, I#1))
    and K in [1..N]
    and ISPERM(A, B)
        .
        .
        .
 H1: all I#1 : INT, I#1 in [K+1..N] -> VALUE_OF_MAX(B, 1, I#1) = B[I#1]
 H2: ISPERM(A, B)
 H3: (K-1) in [1..N]
 H4: K in [1..N]
 H5: K ne 1
-->
 C1: all I#1 : INT,    I#1 in [K..N]
                 ->   VALUE_OF_MAX(EXCHANGE(B,
                        LOCATION_OF_MAX(B, 1, K), K), 1, I#1)
```

```
                    = EXCHANGE(B, LOCATION_OF_MAX(B, 1, K), K)[I#1]
  C2: ISPERM(A, EXCHANGE(B, LOCATION_OF_MAX(B, 1, K), K))
- - - - - - - - - - - - - - - - - - - - - - -
End of path
-----------------------------------------------

Unaffected VCs: EXCHANGE_SORT#1, EXCHANGE_SORT#2, EXCHANGE_SORT#3
```

Three previously-generated VCs (which came from the unchanged paths) were unaffected by the change.

The scheme employed by the DVM for generating only new VCs is discussed in Section 4.7.

**Proof Development**

The incremental aspects of proving formulas both complement and augment the other incremental activities. The term "formula" refers to both VCs and problem-domain facts assumed in proofs. The interface between the DVM and proof system is critical in attaining SID's proof development capabilities. The part of the proof system of interest here is its interactive theorem prover (abbreviated as prover), which is an extension of the program described in Bledsoe and Tyson [75]. The DVM-prover interface is divided according to three main tasks:

1. ***Getting started.*** The DVM invokes the prover with the information necessary to attempt the proof of the designated formula. The DVM retrieves this information from the data base of the assistant.

2. ***Adding to the proof.*** During the proof process the prover may (and typically does) invoke the DVM to obtain additional information. The DVM performs the necessary error checking (does the information exist and can it be used in this context?); then if the request if valid, the DVM retrieves the data directly from the data base (e.g., a basis property or complete specification), or does the appropriate manipulations to get it (e.g., by separating out the unwanted parts of a specification).

3. ***Concluding a proof.*** Upon the completion of a proof, the DVM interprets and stores the results of the attempted proof. Results of a successful proof include a detailed record of the proof that makes explicit steps in derivations along with their justifications.

Keeping this interface in mind will aid in understanding how SID's proof-related activities are accomplished.

***Prove only unproved formulas.*** Changes not only affect VCs, they also affect proofs. This is seen by reconsidering the example of changing the number of arguments to a function. This causes a type mismatch in all of its callers. Suppose that this function is called by assumed properties or specifications used in proofs. Then, proofs using these properties must be redone. As another example, suppose that a previously-used property is incorrect. Revising this property may or may not affect the proofs in which it was used. Section 4.8 discusses the techniques employed by the DVM to determine what proofs (if any) need to be redone when changes affect properties used in proofs.

After the DVM applies the methods of Section 4.8 to identify what proofs need to be redone, the remaining problem is how to retain still-valid parts of these proofs. The DVM supplies the prover with the relevant proof records that need to be at least partially redone and the revised properties. The prover tries to reprove the affected subgoals while ensuring that the proofs are consistent with other parts of the original proof. The proof record is used to supply the necessary context. For example, the prover must be aware of all substitutions made in proofs and also the context in which they were made because substitution conflicts may invalidate previous deductions. The prover currently accepts reproofs when substitutions made in a subgoal's new proof subsume those made in its previous proof.

This completes the linking of changes in the design to proofs. When design changes affect VCs, the methods of Section 4.7 keep intact still-valid ones along with their proofs. When design changes affect properties used in proofs, the two-step process just described is employed to keep intact still-valid proofs. The methods of Section 4.8 separate out all affected proofs, then proofs of their affected subgoals are redone via the prover.

***Proving parts of formulas.*** The prover decomposes formulas into a series of subgoals, each of which can be proved, deferred until later, or assumed with justification. This allows proofs to be done piecewise rather than all at once. For example, the user can do part of a proof, then temporarily discontinue it, then resume it later without having to redo any subproofs that remain valid. Providing for this kind of evolutionary proving is important because difficult propositions that cannot be established automatically are often more conveniently (or perhaps out of necessity!) proved over a period of time.

The ability to defer or assume subgoals is a valuable addition to one's proof repertoire. The defer command is useful when the user wants to think more about what is needed to complete the current proof and go on to others, yet retain already-completed parts of the proof. The assume command is useful when the user sees the obvious truth of a formula and wishes to terminate its automated proof and when the prover is unable to draw the inferences necessary to complete the proof, but the user is able to prove it by hand.

These features complicate part 3 of the DVM-prover interface. The prover, upon concluding the proof process, must convey to the DVM what happened. The information transmitted varies in amount and kind according to what transpired during the proof. Examples of what the DVM needs to know are: what has been proved, what subgoals are assumed, what subgoals are deferred, what their names are, what needs to be saved to resume deferred proofs, and what properties were used where. The DVM assimilates all this information to determine things like the status of the attempted proof and the impact of this status on the overall problem.

Part 1 of the interface also becomes more complex. It must now initiate proofs of deferred subgoals as well as original proofs and reproofs. As with reproofs, the DVM brings together the needed information that was previously collected by the prover. Fitting proofs of deferred subgoals into the overall proof structures is much like the problem of reproving subgoals. The only difference is that deferred subgoals do not have any prior substitutions. The prover currently handles deferred proofs just like reproofs. For the subsumption test to succeed, proofs of deferred subgoals cannot yield any substitutions that need to be used elsewhere in the overall proof.

***Adding new facts.*** Instead of adding all known problem-domain facts (or *basis properties*) at the beginning of proofs, individual facts are added when needed during proofs. Needed facts may already be part of the collection on known facts or added anew. Thus, the user can gradually build up a body of facts that describe a particular domain, and selectively apply individual ones. Adding facts only when needed has long been the policy of the prover [Bledsoe 75]. The interfaces that incorporate this capability into the overall system are new.

The scenario for adding previously-undefined facts to proofs is as follows.

```
Prover-> Use Lemma

Enter lemma . . .
* ALL L:INT, ALL P:INT_ARRAY,
*       L IN [1..N] -> VALUE_OF_MAX(P,L,L) = P[L];
Lemma added . . . Its name is LEMMA*1
```

"Prover->" is the prover's prompt and "*" is the translator's prompt. The typed lemma is parsed and type-checked within the context of the overall problem. Value_of_max, for example, is checked for the correct number and types of arguments. SID assigns the name LEMMA*1 for subsequent reference. The user next requests a display of the current theorem.

```
Prover-> Print Theorem

 H1. L$ in [1..N] -> VALUE_OF_MAX(P$, L$, L$) = P$[L$]
 H2. K=1
-->
 C1. VALUE_OF_MAX(B, 1, 1) = B[1]
```

LEMMA•1 has been skolemized and added as hypothesis H1. Identifiers containing the symbol "$" are skolem variables.

Transparent to the user in this scenario, as in many others, is the large amount of intra-system interfacing needed to carry out this request. Adding a new lemma is a two-step process: entering the lemma into the system and then skolemizing it for addition to the current theorem. Accomplishing these tasks requires several interactions among the prover, DVM, translator, and data base. Reasons for these interactions (all of which are coordinated by the DVM) include the need to access and update the symbol table for type checking purposes and the need to name and store the lemma in the data base for subsequent reference. Figure 3-8 illustrates. Arrows indicate flow of control.

Bringing already-defined facts to bear on a proof is straightforward. The user simply refers to the desired fact by name and it is automatically added to the current hypothesis.

***Expanding deferred specification expansions.*** The notion of adding basis properties to proofs only when needed is carried over to specifications. VCs may contain only references to specifications of called programs instead of the specifications themselves. Then, complete specifications, or parts of specifications, are expanded automatically or interactively as needed during the proof.

The motivation for this is the need to limit the size of VCs. VCs are often quite large and bulky, hindering effective user analysis. For example, some VCs that arise from concurrency in the communications processing example in Appendix A are 50 to 100 lines long. Limiting their size significantly aids the interactive proving process. Therefore, SID defers the expansion of specifications from called programs in both specifications and executable code until the actual proof has begun. (SID also addresses this problem by simplifying VCs as they are generated.) This tends to reduce the size of VCs, enhance readability, and keep them in terms of user-defined abstractions.

Deferred specification expansions can be expanded as either uninstantiated hypotheses or instantiated definitions. The function IsPerm is used in illustrating how specifications are added as additional hypotheses.

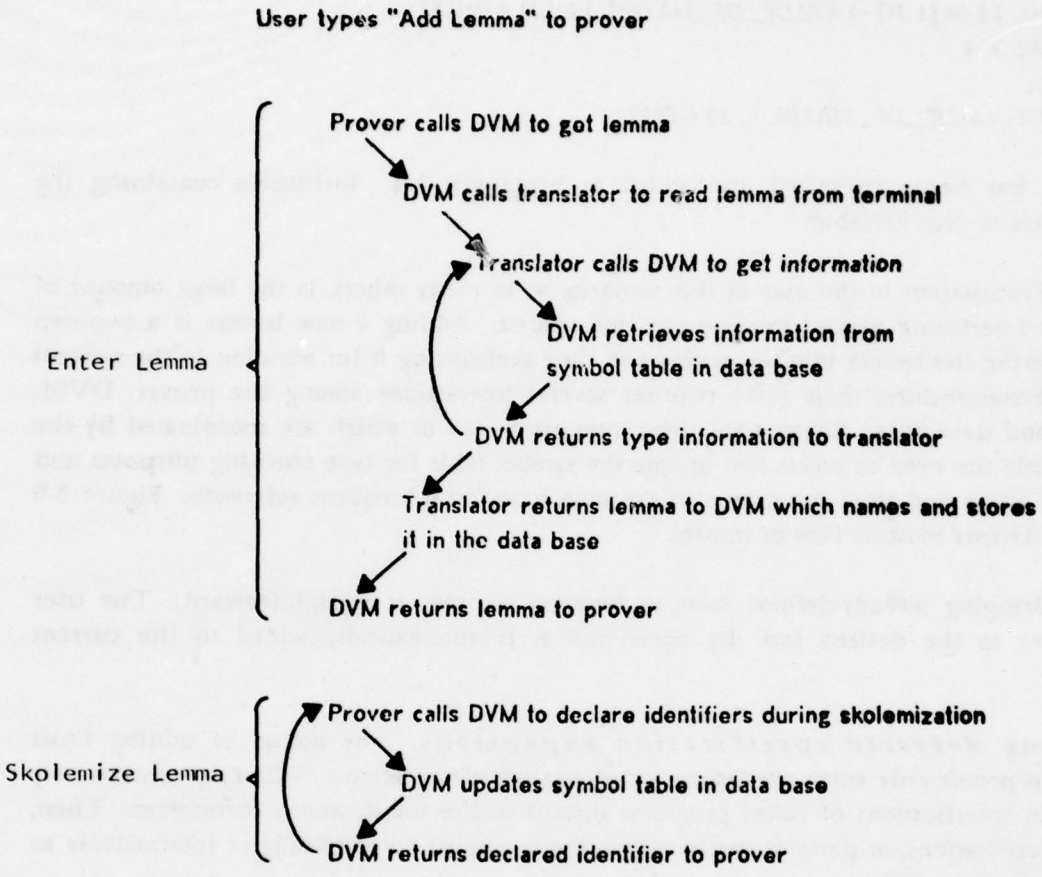



Figure 3-8. Interface structure for adding new lemmas during proofs

```
function IsPerm(X,Y:int_array) : boolean =
begin
  exit (assume
          (all Z:int_array, IsPerm(Z,Z))
        and (all Z:int_array,
               IsPerm(X,Z) and IsPerm(Z,Y) -> IsPerm(X,Y)));
end;
```

Suppose we are proving the following theorem

```
Prover-> Print Theorem

  H1. N in [K..POSINF]
  H2. K in [2..N]
  H3. I in [K+1..N] -> VALUE_OF_MAX(B, 1, I) = B[I]
  H4. ISPERM(A, B)
  H5. K ne 1
-->
  C1. ISPERM(A, EXCHANGE(B, LOCATION_OF_MAX(B, 1, K), K))
```

The proof requires the second exit property of IsPerm. So the user types

```
Prover-> Use IsPerm 2
```

which causes the desired property to be skolemized and added to the theorem.

```
Prover-> Print Theorem

  H1. ISPERM(X$, Z$) and ISPERM(Z$, Y$) -> ISPERM(X$, Y$)
  H2. N in [K..POSINF]
  H3. K in [2..N]
  H4. I in [K+1..N] -> VALUE_OF_MAX(B, 1, I) = B[I]
  H5. ISPERM(A, B)
  H6. K ne 1
-->
  C1. ISPERM(A, EXCHANGE(B, LOCATION_OF_MAX(B, 1, K), K))
```

Hypothesis H1, with skolem variables X$, Y$, and Z$, is the second exit property of IsPerm.

For a look at a definition expansion, consider the following function:

```
function IsExchanged(A,B:int_array; I,J:int):boolean =
begin
  exit (assume IsExchanged(A,B,i,j) iff
              (  I in [1..N]  and  J in [1..N]
              and A[I]=B[J] and A[J]=B[I] ) );
end;
```

Let us focus on one hypothesis of a larger theorem.

```
Prover-> Print Theorem
```

```
H1. ISEXCHANGED(B, EXCHANGE(B, LOCATION_OF_MAX(B, 1, K), K),
          LOCATION_OF_MAX(B, 1, K), K)                          (3.3)
    .
    .
    .
```

Replacing IsExchanged by its definition is done by executing the following command.

```
Prover-> Expand IsExchanged
```

Hypothesis H1 is replaced by four hypotheses.

```
Prover-> Print Theorem
```

```
H1. LOCATION_OF_MAX(B,1,K) in [1..N]
H2. K in [1..N]
H3.  EXCHANGE(B, LOCATION_OF_MAX(B, 1, K), K)[K]
   = B[LOCATION_OF_MAX(B, 1, K)]
H4.  EXCHANGE(B, LOCATION_OF_MAX(B, 1, K), K)
       [LOCATION_OF_MAX(B, 1, K)]
   = B[K]
    .
    .
    .
```

Hypotheses H1-H4 are obtained by instantiating the definition of IsExchanged with the arguments in (3.3). That is, substitute B for A, Exchange(B, Location_of_max(B,1,K),K) for B, Location_of_max(B,1,K) for I,and K for J.

Sometimes these expansions are triggered automatically. If an unexpanded reference is the sole hypothesis, it is expanded under the assumption that it is likely to be needed in the

proof. Also, unexpanded references in the conclusion are expanded so they can be proved. These situations often are not present when the proof begins, but arise during the proof as a result of the proof strategy employed.

Specification expansions are not always deferred until proof time. It is sometimes desirable to have the VC generator automatically add the specifications of called programs to VCs, instead of inserting references to the specifications. For example, VCs for establishing entry conditions have the entry condition of called programs fully expanded in their conclusion. Full expansion is also done when proving properties of concurrent processes. An example is the VCs for the top-level network program in Appendix A. This program calls processes whose block specifications are fully expanded in its VCs. This is done because current experience with concurrency proofs using SID indicates that this expansion is nearly always needed.

## 3.4 The System Design

Previous sections in this chapter sketched SID's overall design, described its designer/verifier's assistant, and discussed its incremental capabilities (along with the interactions involved in accomplishing them). This section takes a more detailed look at SID's design, focusing on its important design characteristics and its traditional components.

Figure 3-9 gives a detailed view of SID's organization. (The assistant was broken down in Fig. 3-2.) Solid arrows indicate program calls visible to the user and dashed arrows indicate calls that are transparent to the user. The impact of adopting an integrated, incremental view of design and verification is evident in this diagram.
The methods of Chapter 4, which are implemented in the DVM part of the assistant, encourage:

- ***Centralization of control.*** To perform a particular task, the DVM determines what work needs to be done and invokes the appropriate program, or sequence of programs. Tasks are initiated by the user or by an invoked program.

- ***Centralization of information.*** The DVM controls all data-transmission links in the system. All data is stored in a single data base directly accessible only to the DVM.

There is one exception to the centralization of control principle. For efficiency, the automatic simplifier is called directly to manipulate intermediate data. During VC generation, for example, the simplifier is called every time an assignment statement is traversed. The

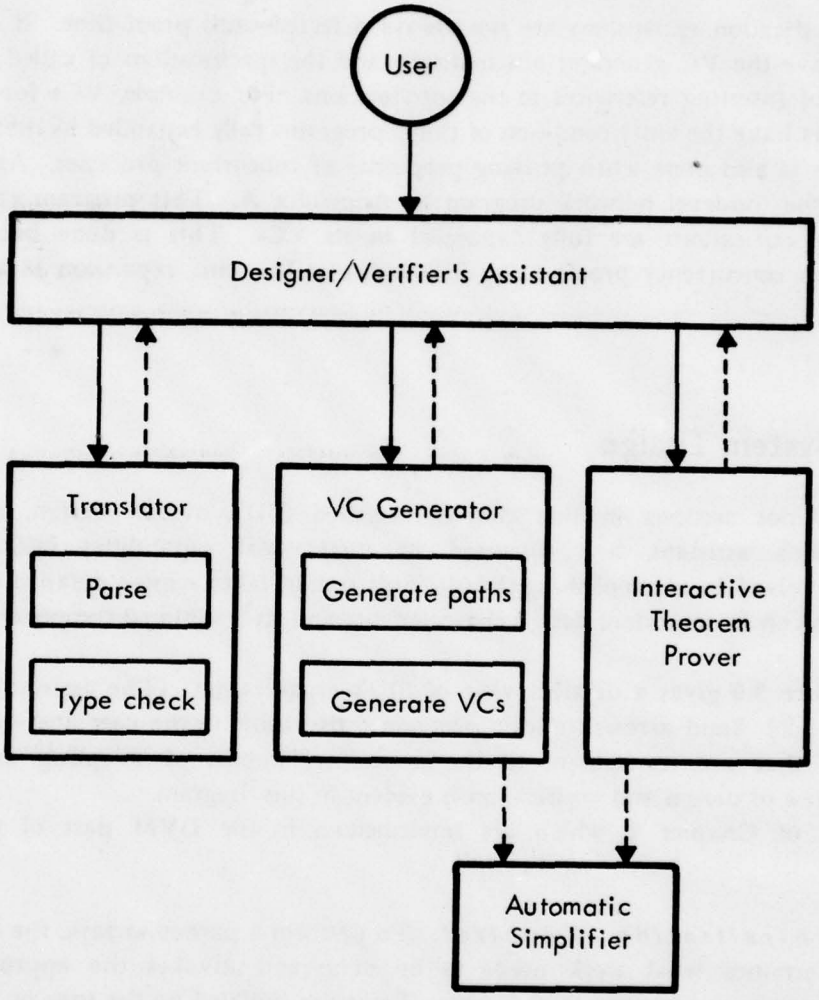



Figure 3-9. Detailed structure of SID

frequency of these calls suggests using a direct interface, which does not require things like data reformatting by the DVM.

Both of these design characteristics have important benefits. Centralizing control enables the methods of Chapter 4 to screen intended operations prior to their execution and thereby prevent things like generating VCs that will contain type conflicts. It also enables the DVM to coordinate diverse tasks. Figure 4-8 illustrated how the DVM coordinates the translator, prover, and data base when new lemmas are added during proofs.

Centralizing information enables the DVM to provide a uniform set of functions for accessing all system data. These functions use primitives provided by each component for manipulating the data they produce. The translator, for example, provides symbol table primitives. Components, therefore, can access information without knowledge of its origin or representation details. For example, the prover frequently needs type information to ensure that only integer-valued variables are entered into its typelist mechanism. The prover obtains this type information without knowledge of the translator or symbol table representation. Centralization of control surfaces here too because the assistant, being aware of context, established the appropriate scope in the symbol table.

The discussion below describes the system components tied together by the assistant -- the translator, VC generator, simplifier, theorem prover, and display package (not in Fig. 3-9). The focus is on what they do and the impact of the system goals on their design.

### Translator

This program -- developed by Wilhelm Burger -- consists of a Gypsy parser and type checker. A code generator is under development. The translator is table driven and applies to programs, specifications, and basis properties. Its parse tables are generated by the BOBSW system [Burger 74]. Diagnostics are given in the event of syntactic or type errors. Processed entities that either have no errors or have only type errors are stored in the assistant's data base for future reference, whereas entities containing syntax errors are not.

The impact of the methods of Chapter 4 on the overall design of the translator is twofold. First, the translator, in addition to having and entry point for parsing and type checking, also has an entry point for type rechecking. All programs, specifications, and basis properties can be rechecked. What entities to recheck and when to recheck them is decided by the methods of Chapter 4. Second, all type checking is done in the context of the overall problem as represented in the assistant's data base.

**Verification Condition Generator**

This program -- developed by Rich Cohen -- serves the twofold purpose of generating VCs and of generating a record of the statements involved in the generation of each VC. The proof methodologies implemented are the conventional inductive assertion method [Floyd 67], and the data abstraction and concurrency proof techniques described in Good [77].

The importance of limiting the size of VCs to enhance user analysis has been discussed. The VC generator addresses this problem in two ways:

- ***Incremental simplification.*** The VC generator uses the automatic simplifier to simplify VCs as they are generated. This eliminates much of the unnecessary clutter and allows mechanical, propositional inferences (sometimes resulting in logical truths) to be made before the VCs are displayed to the user.

- ***Defer specification expansions.*** Only references to some specifications of called programs are inserted in VCs, instead of the specifications themselves. Section 3.3 gives examples and shows how such references are expanded during proofs.

Simplifying during VC generation has potential drawbacks because the relationship between VCs and the statements involved in their generation may be obscured. When a proof attempt fails, this correspondence is needed to determine if there is an inconsistency between the program and its specifications. This is sometimes easier with unsimplified VCs. The VC generator responds to this issue by recording the sequence of statements used in the generation of each VC along with some annotations. The user may have these annotated paths displayed in parallel with VC generation, or simply request them later as needed. The need for these commentaries is amplified in SID because of complications involved in verifying data abstractions and concurrency.

**Automatic Simplifier**

This program simplifies logical, limited relational, and arithmetic expressions. It has a built-in knowledge of the propositional calculus and is fully automatic. The VC generator uses it to simplify VCs as they are generated, and the theorem prover uses it to simplify formulas added during proofs. The philosophy behind the simplifier is to perform efficient, straightforward manipulations without interaction before invoking the more powerful capabilities of the interactive theorem prover.

Since its original use in the verification system of Good, et al. [75], Don Good made extensions it to handle rational division, various symbolic state manipulation capabilities needed to accommodate the VC generator, and built-in Gypsy operators for manipulating things like sequences.

**Interactive Theorem Prover**

The interactive theorem prover is a variation of an already highly developed program originally developed as part of the Automatic Theorem Proving Project at UT Austin (see Bledsoe and Tyson [75]). It is a natural deduction system (i.e., a Gentzen-like system as opposed to a "less natural" system such as resolution) that proves theorems by subgoaling (splitting), matching, and rewriting. Its handling of certain formulas from Presburger arithmetic [Bledsoe 75] and proofs by cases makes it particularly effective in verification [Good, et al. 75]. Mabry Tyson augmented this prover in several ways for use in SID.

***User interface.*** The prover's command structure was reworked for compatibility with the assistant's. The prover and assistant use the same command scanner (also written by Mabry Tyson) so that user commands are handled uniformly at all interaction points. Unlike the assistant, the prover does not offer suggestions, nor does it answer English queries.

***Additional information.*** Facilities for handling deferred specifications and using properties from the assistant's data base were added. Examples of both are in Section 3.3.

***Proof presentation.*** Since successful proofs may depend upon properties that incorrectly describe the domain, easily-understandable records of proofs (displaying how properties are used) are important as documentation for verifications to be credible. Seeing how properties are used is important, for example, because individual properties that appear to accurately characterize the domain may subtly interact during a proof to cause unintended inferences. Thus, the prover collects a detailed record of proofs that makes explicit both built-in and user-supplied assumptions. An initial version of a dual-mode display facility is available for viewing these proofs. The user has the option of interactively directing presentations by requesting the desired amount of detail at selected steps, or of having a completely automatic presentation by specifying initially the amount of detail desired throughout.

***Proof interruption.*** The user can direct the prover to abort, assume, or defer proofs. The abort command is useful, for example, when the user sees that he is trying to prove a false theorem and wishes to terminate the search. Section 3.3 discusses the assume and defer capabilities.

**Display Package**

Dwight Hare developed a set of routines for formatted printing of data, including programs, specifications, and basis properties. Data is translated from its internal prefix form to formatted infix. For example, the lemma represented in prefix as

```
(ALL D
    (TYPREF INT_ARRAY)
    (ALL R
        (TYPREF INT)
        (ALL S
            (TYPREF INT)
            (IMP (AND (AND (IN R (RANGE 1 N)) (IN S (RANGE 1 N)))
                      (LT (FUNCTION VALUE_OF_MAX D R S)
                          (ARRAY D (PLUS S 1))))
                 (EQNUM (FUNCTION VALUE_OF_MAX D R (PLUS S 1))
                        (ARRAY D (PLUS S 1)))))))
```

is displayed to the user as

```
all D : INT_ARRAY,
  all R, S : INT,       R in [1..N] and S in [1..N]
                    and VALUE_OF_MAX(D, R, S) < D[S+1]
                  -> VALUE_OF_MAX(D, R, S+1) = D[S+1]
```

Options are available for varying indentations and mixing upper and lower case. This example has keywords in lower case. Several components, especially the assistant, interface with this package.

## 3.5 Implementation Details

SID is entirely Lisp based, completely compiled, and runs on a PDP-10 computer under TOPS at The University of Texas at Austin. Most of the system is written in Reduce [Hearn 71]; the rest is written directly in UCI Lisp.

When designing and verifying programs like the sorting example or the message switching network in Appendix A, SID occupies approximately 186K of core. This amount includes the Lisp system; the Reduce translator; all of the programs, tables, and data; and enough working free storage to handle complex manipulations. Figure 3-10 summarizes how memory is used. Figure 3-11 details the memory requirements of SID's compiled code, breaking

down memory into binary program space (area for compiled functions and arrays), free storage (areas for Lisp nodes), and full word space (area for print names and numbers). Figure 3-12 summarizes the memory used by the supporting software.

| | |
|---|---|
| SID's code | 108 |
| Software support | 48 |
| Working Storage | 15 |
| Data base for sort (network) | 10 (15) |
| TOTAL | 181K (186K) |

Figure 3-10. Memory requirements for running SID in thousands of PDP-10 words

| | |
|---|---|
| Binary program space | |
| assistant | 12 |
| translator (includes tables) | 11 |
| VC generator | 6 |
| interactive theorem prover | 16 |
| automatic simplifier | 7 |
| display package | 6 |
| Free storage | 24 |
| Full word space | 8 |
| Global initializations | 18 |
| TOTAL | 108K |

Figure 3-11. Memory requirements for SID's compiled code

| | |
|---|---|
| Lisp system | 35 |
| Reduce (without algebraic package) | 13 |
| TOTAL | 48K |

Figure 3-12. Memory requirements for supporting software

# CHAPTER 4

# METHODOLOGY FOR RESPONDING TO CHANGES

This chapter first overviews the methodology, then details the algorithms and the underlying data base they manipulate. The term "program unit" is used to refer to a program and its specifications.

## 4.1 Preview of the Methodology

This section gives a preliminary overview of the methods for dealing with changes to programs, specifications, and basis properties. Both the general approach and important characteristics of the methods are discussed.

### Key Ideas

Designing and verifying programs involves developing a highly-interrelated collection of several kinds of information, including programs, specifications, VCs, basis properties, and proofs. Initially, the methods are best understood by considering only some of this information and a few ways in which it is related. The sorting program of Section 1.2 again is used for illustrative purposes.

Figure 4-1 gives the final version of the sorting program accompanied by the lemmas used in its verification. The discussion below refers to the line numbers to the left of the listing, which are not a part of the program or lemmas. From this information, the methods build the cyclic graph structure in Fig. 4-2 to describe calling relationships. Arrows in Fig. 4-2 indicate direct calls in Fig. 4-1. For example, Exchange_sort calls itself on line 9, Location_of_max on line 19, Value_of_max on line 10, Exchange on line 19, and IsPerm on line 11. Figure 4-2 omits data definitions because they are handled differently from functions.

Suppose that the definition of Location_of_max is altered by interchanging lines 48 and 49. What are the effects of this change? Figure 4-3 gives a first approximation of the answer, excluding those functions that do not directly or indirectly call Location_of_max from consideration. This approximation is refined by observing that Exchange_sort cannot be affected because lines 48 and 49 are in a part of Location_of_max that is not visible to its

```
1     const n:int = pending;
2
3     type int_array = array ([1..n]) of int;
4
5     function Exchange_sort(A:int_array):int_array =
6     begin
7       entry N ge 1;
8       exit (all I:int,
9               I in [1..N] ->   Exchange_sort(A)[I]
10                                = Value_of_max(Exchange_sort(A),1,I))
11            and IsPerm(A, Exchange_sort(A));
12      var B:int_array := A;
13      var K:int := N;
14      keep  K in [1..N];
15      loop
16        assert (all I:int, I in [k+1..N] -> B[I] = Value_of_max(B,1,I))
17          and K in [1..N]  and  IsPerm(A,B);
18        if K = 1 then leave end;
19        B := Exchange(B,Location_of_max(B,1,K),K);
20        k := K-1;
21      end;
22      result := B;
23    end;
24
25
26    function Value_of_max(A:int_array; I,J:int):int =
27    begin
28      exit ( assume ( all k:int,
29          k in [I..J] and I in [1..N] and J in [1..N]
30            -> A[k] le Value_of_max(A,I,J) )
31        and ( all I,m:int,
32          I in [I..J] and m in [I..J] and I in [1..N] and J in [1..N]
33            -> Value_of_max(Exchange(A,I,m),I,J)  =
34              Value_of_max(A,I,J) ) );
35    end;
36
```

Figure 4-1. Final version of sorting program and lemmas used in its verification
(continued on next page)

```
37    function Location_of_max(A:int_array; I,J:int):int =
38    begin
39      entry I in [1..N] and J in [1..N] and I le J;
40      exit Location_of_max(A,I,J) in [I..J]
41        and A[Location_of_max(A,I,J)] = Value_of_max(A,I,J);
42      var K:int := I;
43      keep I le J and I in [1..N] and J in [1..N] and K in [1..N];
44      result := I;
45      loop
46        assert A[result] = Value_of_max(A,I,K)
47          and result in [I..K] and K in [I..J];
48        if K ge J then leave end;
49        K := K+1;
50        if A[result] < A[K] then result := K end;
51      end;
52    end;
53
54    function Exchange(A:int_array; I,J:int):int_array =
55    begin
56      entry I in [1..N] and J in [1..N];
57      exit  (all k:[1..N],
58              k ne I and k ne J -> Exchange(A,I,J)[k] = A[k] )
59        and IsExchanged(A,Exchange(A,I,J),I,J)
60        and IsPerm(A,Exchange(A,I,J));
61      result := A;
62      result[I] := A[J];
63      result[J] := A[I];
64    end;
65
66    function IsExchanged(A,B:int_array; I,J:int):boolean =
67    begin
68      exit (assume IsExchanged(A,B,i,j) iff
69                  (  I in [1..N]  and  J in [1..N]
70                  and A[I]=B[J] and A[J]=B[I] ) );
71    end;
72
```

Figure 4-1. (cont'd) Final version of sorting program and lemmas used in its verification (cont'd. on next page)

```
73    function IsPerm(X,Y:int_array) : boolean =
74    begin
75      exit (assume
76              (all Z:int_array, IsPerm(Z,Z))
77            and (all Z:int_array,
78                  IsPerm(X,Z) and IsPerm(Z,Y) -> IsPerm(X,Y)));
79    end;
80
81
82      LEMMA#1 is
83        all L : INT,
84          all P : INT_ARRAY, L in [1..N] -> VALUE_OF_MAX(P, L, L) = P[L]
85
86      LEMMA#2 is
87        all D : INT_ARRAY,
88          all R, S : INT,        R in [1..N] and S in [1..N]
89                              and VALUE_OF_MAX(D, R, S) < D[S+1]
90                           -> VALUE_OF_MAX(D, R, S+1) = D[S+1]
91
92      LEMMA#3 is
93        all D#1 : INT_ARRAY,
94          all R#1, S#1 : INT,        R#1 in [1..N] and S#1 in [1..N]
95                              and VALUE_OF_MAX(D#1, R#1, S#1) ge D#1[S#1+1]
96                           ->   VALUE_OF_MAX(D#1, R#1, S#1+1)
97                              = VALUE_OF_MAX(D#1, R#1, S#1)
98
99      LEMMA#4 is
100       all L, M : INT,
101         all P : INT_ARRAY,    L in [1..N] and M in [1..N]
102                           -> ISPERM(P, ALPHA(ALPHA(P, L, P[M]), M, P[L]))
103
```

Figure 4-1. (cont'd.) Final version of sorting program and lemmas used in its verification

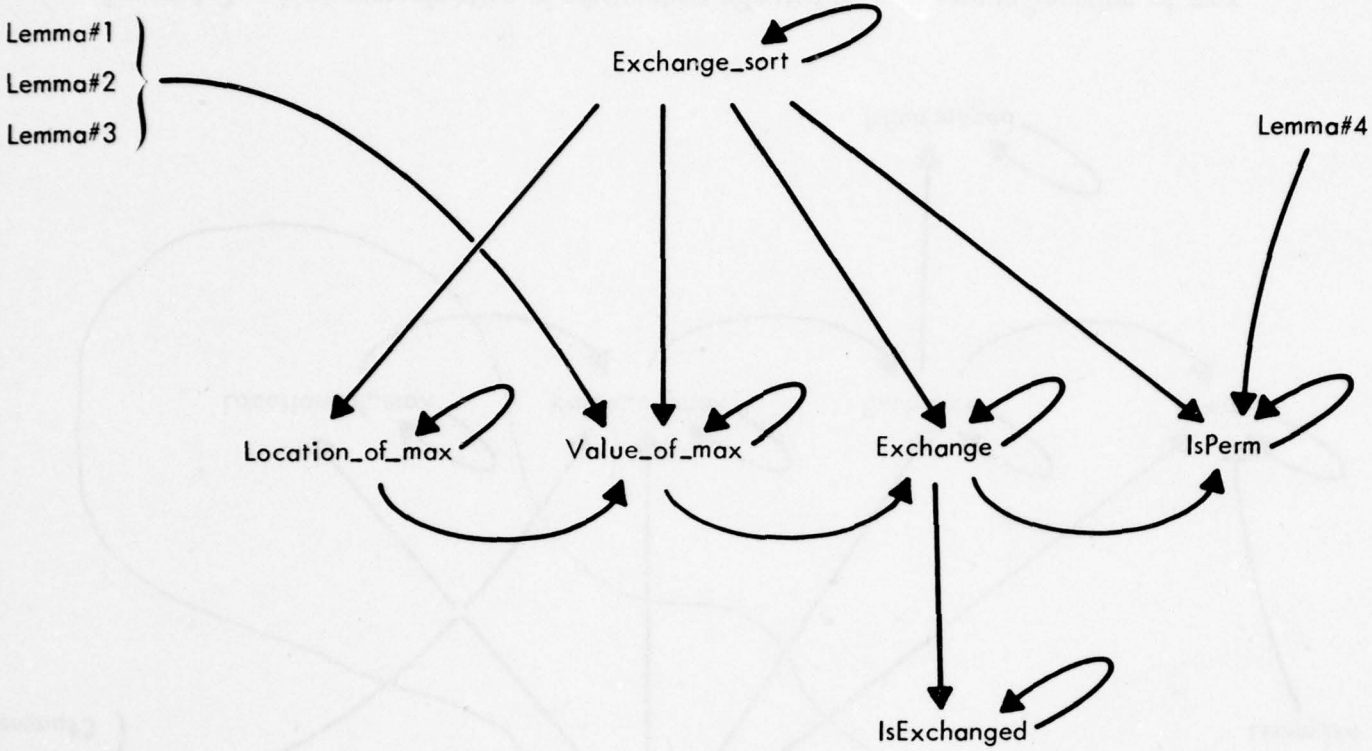



Figure 4-2. Calling relationships in Fig. 4-1

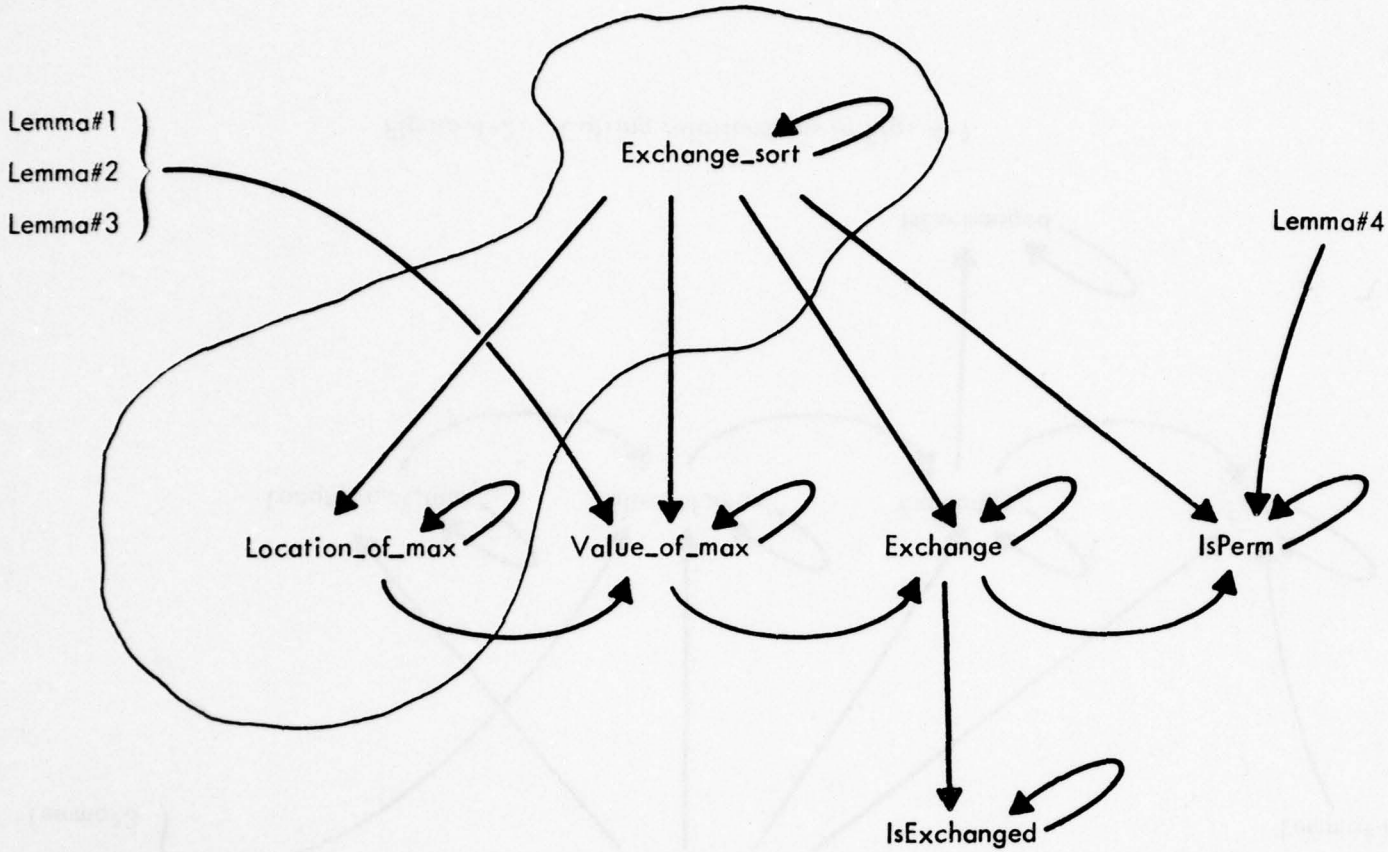



Figure 4-3. First approximation of substructure affected by a change to Location_of_max

callers. But Exchange_sort can be affected by a change to (or changes that affect) the header of Location_of_max on line 37. From these simple observations comes a fundamental principle -- changes have global effects only through externally-visible parts of program units; conversely, changes to other parts have only local effects.†

Distinguishing between external and internal parts of program units is usually only the first step in determining the effects of changes. Suppose that the header of IsExchanged on line 66 is changed to

```
function IsExchanged(A,B:int_array; I:int):boolean =
```

The argument J has been deleted, changing the number of arguments to IsExchanged from four to three. Figure 4-4 gives a first approximation of the affected substructure, excluding IsPerm and Lemma4 from consideration. Applying the same "internal/external" principle as before yields a duplicate of Fig. 4-4. The initial approximation is not refined. One effect of changing the number of arguments to IsExchanged is a type mismatch in the exit specification of Exchange (line 59), which can be exported for use in the proof of callers of Exchange. This exit specification can be exported to Exchange_sort via several routes, including an indirect one through Value_of_max (because the exit specifications of Value_of_max call Exchange on line 33 and Exchange_sort calls Value_of_max on line 10).

There are several more ways in which the propagation of the effects of changes are constrained. One way is by knowledge of where properties have been used. Figure 4-5 describes where the exit specifications of Exchange and IsExchanged have been used. These descriptions enable the methods to reduce the substructures in Fig. 4-4 to the one in Fig. 4-6. Other constraints detailed later include rules for deciding when changes to previously-used specifications have no effects.

It is not enough to know only what programs are affected by changing the number of arguments to IsExchanged. How they are affected is also important. Two general kinds of consistency are introduced for classifying the effects of changes. *Design consistency* refers to the syntactic and type consistency of programs, specifications, and basis properties. *Verification consistency* refers to the correctness of VCs (e.g., that VCs contain no type errors and reflect changes in the associated programs and specifications) and to the use of specifications and basis properties in proofs (e.g., that no proofs depend on design inconsistent properties and that the needed logical relationships hold between intermediate and final versions of properties used in proofs). Applying these classifications to the programs in Fig. 4-6 yields Fig. 4-7. IsExchanged is design inconsistent, for example, because of a mismatch in the number of arguments between its recursive call on line 68 and its changed header. Exchange has a similar type mismatch on

---

† Changes to external parts of units may also have local effects, as will be shown later.

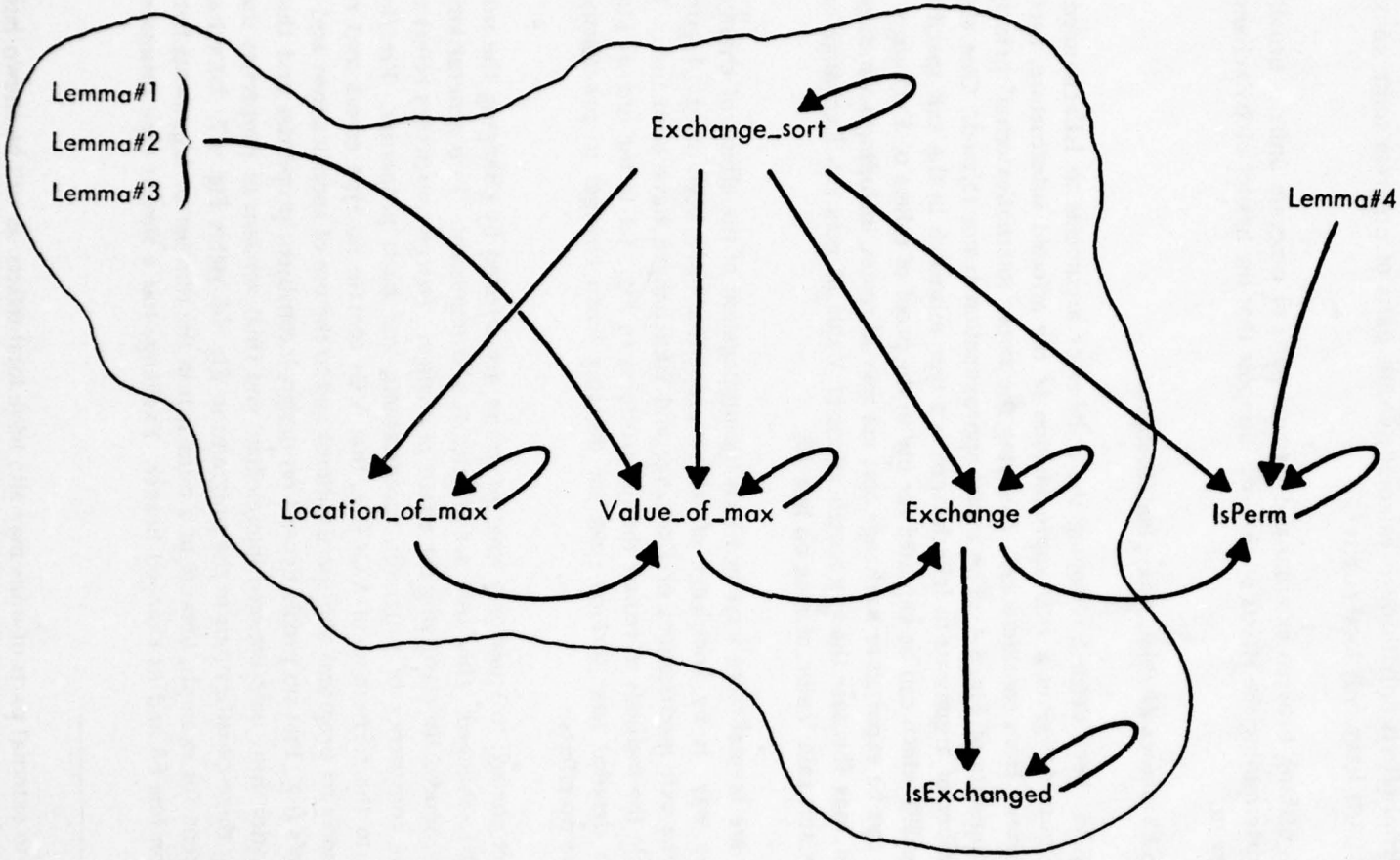
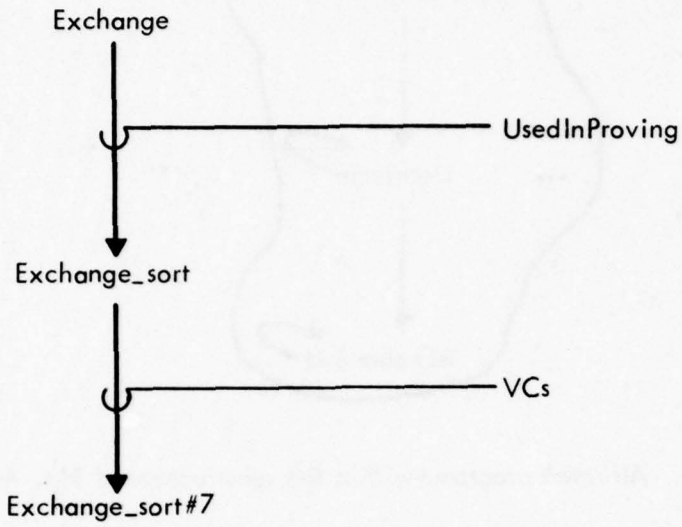
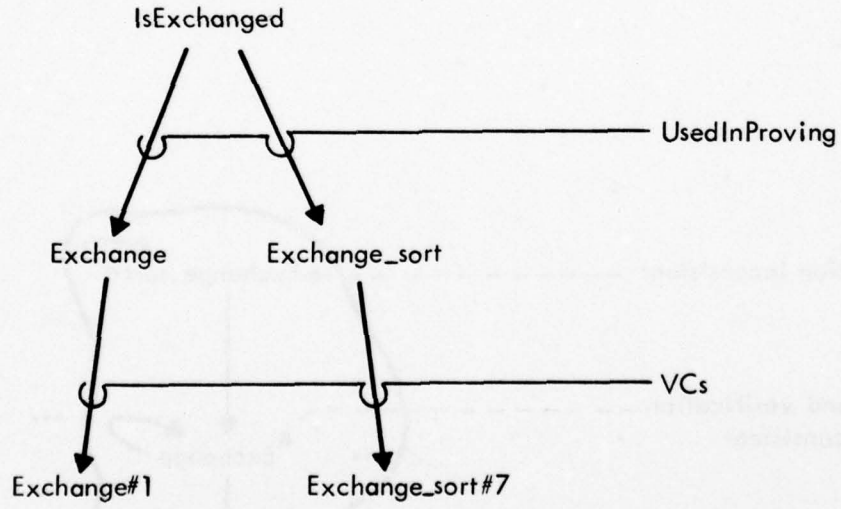



Figure 4-4. First approximation of substructure affected by a change to IsExchanged




(a)




(b)

Figure 4-5. Additional descriptions showing where the exit specifications of Exchange and IsExchanged have been used

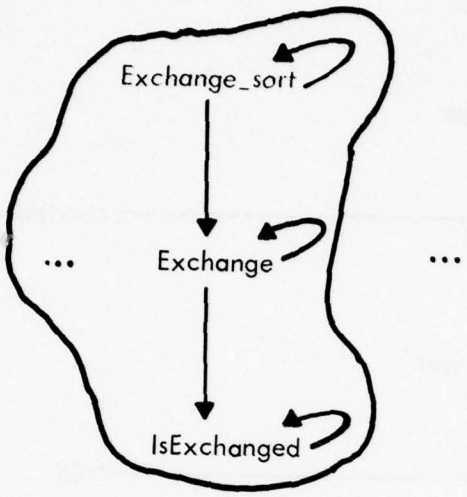



Figure 4-6. *Affected programs within the substructure of Fig. 4-4*

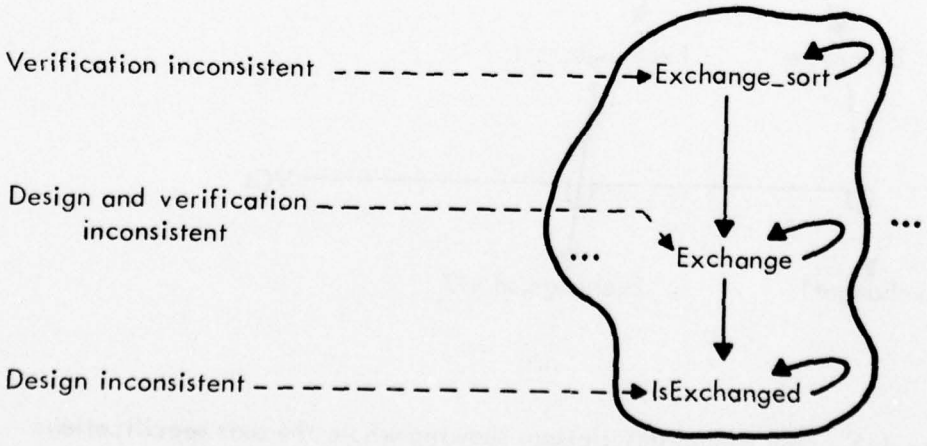



Figure 4-7. Kinds of effects on the programs of Fig. 4-6

line 59 and also a verification inconsistency because this mismatch is in Exchange•1, as shown in Fig. 4-5(b).† Only the verification of Exchange_sort is affected. This is due to its use of the exit specifications of Exchange and IsExchanged (which now have a type mismatch) in the proof of Exchange_sort•7, as shown in Fig. 4-5.

In addition to determining the effects of changes, the methodology must also ensure that all inconsistencies are eventually resolved and enforce any constraints intermediate ones might imply. This involves things like not allowing VCs to be regenerated for Exchange in Fig. 4-7 or its second exit specification to be used in proofs until the type mismatch on line 59 is corrected.

In summary, the effects of changes are determined by exploiting useful constraints on how objects in the domain interact with one another. Previous examples illustrated the utility of several constraints -- calling relationships, external and internal parts of program units, and uses of specifications -- in curtailing the propagation of the effects of changes. The less the propagation, the more previous work that is kept intact. These and other important constraints are explained in detail in the remaining sections in this chapter.

## Characteristics of the Methodology

***Allows temporary inconsistencies.*** Temporary design and verification inconsistencies are allowed in intermediate problem states. This affords several flexibilities. For example, top-down design is enhanced by allowing calls to as yet undefined programs. These called programs can be (partially) defined whenever convenient, thereby eliminating prior design inconsistencies. A common verification inconsistency arises when proofs depend on specifications that now, as a result of a change, contain a type error. Later, the user may wish to undo the change, reinstating the validity of the proofs.

***Applies to any of several strategies.*** The methods apply to any of several program design and verification strategies. Programs and specifications are constructed using a top-down, bottom-up, or mixed design strategy and the verification is carried out in parallel or any desired order. Changes are made whenever convenient.

***Adapts to a class of languages.*** Although applied here to Gypsy, the methods are intended for use with other Pascal-like programming languages and their related proof methods. A set of language-dependent functions is coded for the language of interest. These

---

† Additional knowledge is used to determine that IsExchanged does not have a verification inconsistency. Namely, it is for specification purposes only and, therefore, does not have VCs.

functions are called by algorithms which remain essentially intact for different languages and proof methods. Also, some table-driven algorithms require a description of certain interrelationships between specifications and proof methods.

***Delays consistency checking until necessary.*** The scheme for resolving inconsistencies is based on the following premise -- trying to resolve inconsistencies when they occur is often fruitless since a sequence of changes is often made, the last of which resolves all intermediate inconsistencies. The methods, therefore, simply note that consistency checks need to be made, then perform the needed checks in response to user-initiated operations. That is, it automatically makes those checks necessary to guarantee a consistent state change.

***Generally avoids case-by-case analysis.*** The methods adopt the policy of treating changes to externally visible parts of programs as if they always affect callers, rather than analyzing the exact change on a case-by-case basis. This strategy is much simpler than doing case-by-case examinations. For example, it simplifies the interface between the methods and the type checking program. Rather than communicating via sophisticated error-classification schemes, a boolean-valued interface is all that is needed. It appears, however, that case-by-case analysis is useful in some situations, especially with the aid of a knowledgeable editor such as the one in Yonke [75]. Identifying the classes of changes that should be dealt with individually is an area for future research.

***Some uses of case-by-case analysis.*** Changes to basis properties and specifications are dealt with on an individual basis. Although the above discussion ruled out this approach for the analysis of changes to programs, it is viable for isolating the exact impact of changes to formulas. Typically, such changes may affect several verifications. In practice, it can often be shown that at least parts of the verification are unaffected -- thereby avoiding unnecessary reverification. The methods detect such situations by making logical "comparisons" between a formula and its revised version when neither contains a type error.

***Assumptions.*** The methods assume that type checking of programs, specifications, and basis properties is done at compile time and that programs do not have side effects except through explicitly declared var parameters. Gypsy eliminates side effects by disallowing non-local variables and by controlling interactions among concurrent processes. Concurrent processes communicate by performing send and receive operations on buffers.

**Notes to the Reader**

While studying the formalization given in this chapter, the reader should keep in mind that although algorithms are discussed separately by topic, they are highly interrelated and should, therefore, be viewed collectively. Also, efficiency is sacrificed for simplicity throughout. Before explaining the algorithms, Section 4.2 first describes (as background) the

underlying data-base structure used by the methods and notation to be used throughout. Then, Sections 4.3 - 4.7 explain the methodology as it applies to conventional functions and procedures, and to concurrent process definitions in Gypsy. Specific algorithms and the data structures they manipulate (in addition to the data base) are detailed. To simplify the presentation, basis properties are considered separately in Section 4.8. Section 4.9, building on the established framework, explains how types and data abstraction mechanisms are handled.

The methods are formulated to allow for complete expansion of specifications from called programs during VC generation. To illustrate, imagine a hierarchy of specifications in which a lower-level specification contains a type conflict. The formulation adopts the conservative view of expanding all these specifications in VCs, even though it may be possible to do all proofs without expanding specifications to a depth that reveals the design inconsistency. Section 4.10 describes how the methods were tailored for use in SID to account for this possibility.

Phrases like "X contains semantic errors" and "X is semantically inconsistent" refer to those aspects of semantics handled by type-checking programs in compilers. This includes resolving external references and ensuring that program calls are set up properly.

The convention of designating changed data with a prime symbol is adopted throughout. For example, a revised version of unit X is referred to as X'.

## 4.2 Data-base System

This section describes the data base of facts (operated on by programs described later in this chapter) and its accompanying access mechanisms. Then, notational abbreviations for expressing data-base calls are introduced.

### The Structure for Holding Data

Associated with every program unit and basis property is a collection of related data. A sample is shown in Fig. 4-8, which gives data stored about Location_of_max and illustrates how the data is structured.

SID represents such structures as hierarchically-structured lists of attribute/value pairs. Attributes are atoms and values are atoms, lists of atoms, or general s-expressions. The important descendants possible for each program unit are described, with subsequent discussions clarifying their meaning and use. Omitted here are data-base entries for describing data definitions (see Section 4-9) and what information is backed up when changes occur (see

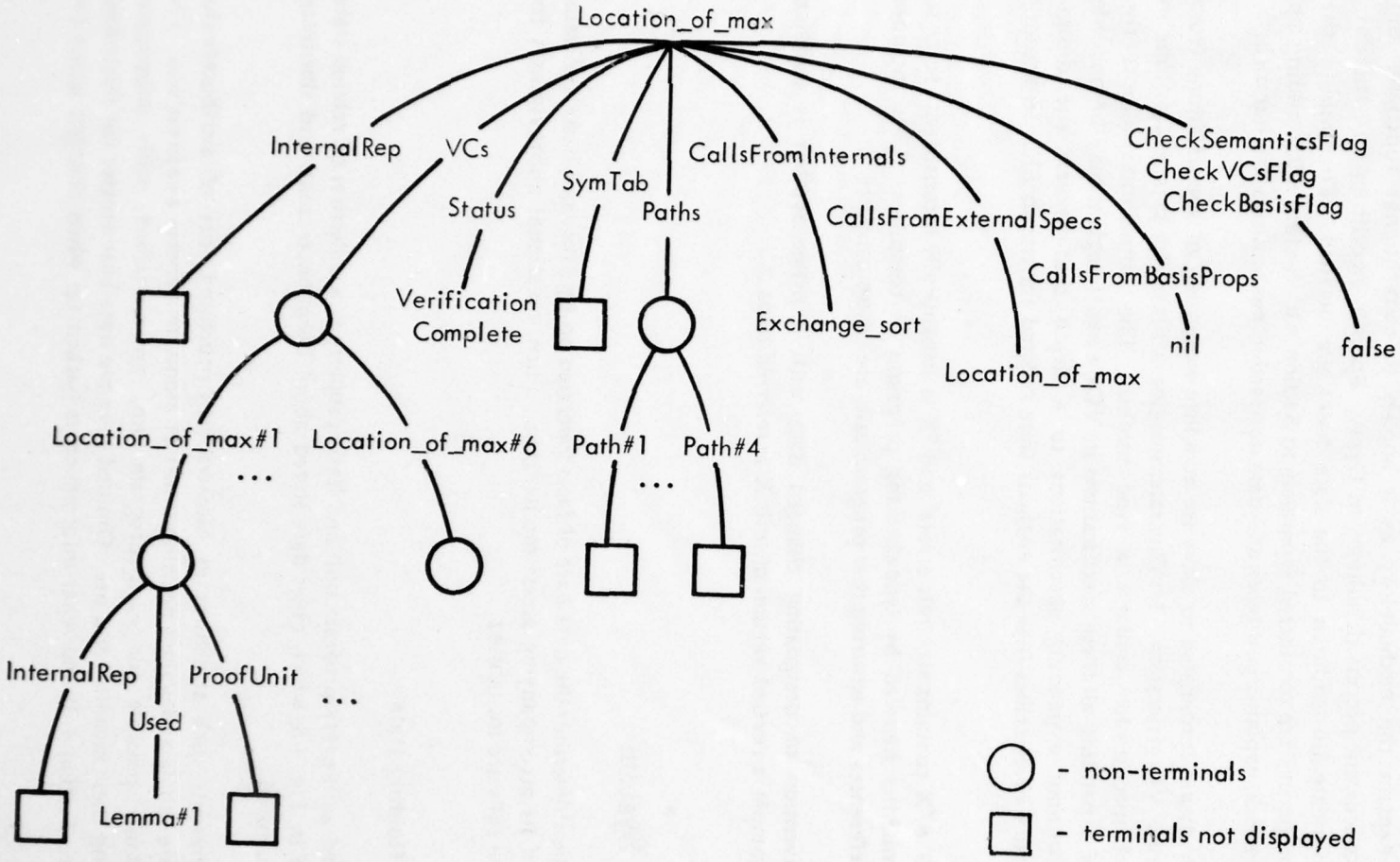



Figure 4-8. Data about Location_of_max stored in the data base

Sections 4-4 and 4-6). Listed below, in boldface, are top-level attributes for program units, accompanied by a description of their values. The atom UnitName refers to the name of the program unit being described.

**InternalRep** is the internal representation of UnitName initially generated by the translator. Each program unit is stored as a single entity and programs extract different parts when needed. Terminals are given default assignments until specific data is available. The default value of "completely undefined" is assigned here if there is data about UnitName (e.g., that it is called by a lemma) before any part of its definition is successfully parsed.

**VCs** consists of VCname:value pairs, where VCname is the system-generated name of a VC and value is a list of three possible pairs whose attributes are -- InternalRep, Used, and ProofUnit. InternalRep references the internal form of the VC; Used has as a value the names of the basis properties used in the proof of VCname; and ProofUnit is the proof structure returned by the theorem prover after attempting to prove VCname.

**Status** is the current status of UnitName. Some possible values of Status are "to be VC generated", "to be proved", and "partially proved". Actually, many of the direct descendants of UnitName have status values for ease of analysis, updating, and responding to user requests. Status possibilities for a VC include "proved in theorem prover" and "proved except for one deferred subgoal".

**SymTab** is the local symbol table for UnitName defined by the type checker.

**Paths** consists of PathName:value pairs, where PathName is the name of a program path whose value is a path produced by the VC generator. Paths and VCs are related by cross references.

**CallsFromInternals** contains the names of program units that call UnitName in their executable text or internal specifications.

**CallsFromExternalSpecs** contains the names of program units that call UnitName in their external specifications.

**CallsFromBasisProps** contains the names of the basis properties that call UnitName. The default value of nil in Fig. 4-8 indicates that no basis properties call Location_of_max.

**CheckSemanticsFlag** is a boolean flag that indicates whether UnitName potentially contains type errors. The value of false in Fig. 4-8 indicates that Location_of_max does not contain type errors.

**CheckVCsFlag** is a boolean flag that indicates whether the VCs for UnitName are potentially incorrect.

**CheckBasisFlag** is a flag/value that indicates if the verification of UnitName is potentially invalid due to a change to, or an inconsistency in, a basis property. The value of this flag, when set, is the names of those basis properties that invalidate UnitName's verification. These properties are associated with particular VCs via the Used attribute under VCs.

Basis properties added by the user to complete a proof are integrated with programs and their specifications in the data base. An example of how they are stored is seen in Fig. 4-9. Basis properties are linked to the verification by their UsedIn attribute and by the CheckBasisFlag for program units, and to the design by the CallsFromBasisProps attribute for program units.

**InternalRep** is the internal representation of the basis property.

**UsedIn** is a list containing each UnitName employing this basis property in its verification.

**CheckSemanticsFlag** is a boolean flag as defined above.

**NotUsableFlag** is a boolean flag that indicates if the property can currently be used in proofs.

**Data Manipulation**

There are three types of data base calls -- insertion, retrieval, and deletion. Insertions into the data base are made by the call

Insert(key,attribute, . . . , attribute,value)

Insert is used both to add new data items and to replace existing ones. Insertions involve two operations: storing the new data item (replacing the old data item, if necessary) and connecting

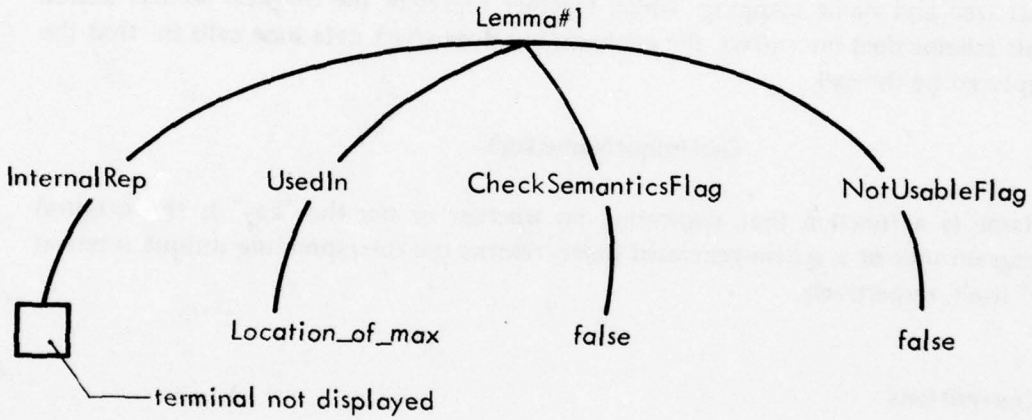



Figure 4.9 Data about Lemma#1 stored in the data base

the new data item to its parent. The key field contains the name of a program unit or a basis property, attributes identify the desired descendant of the key, and the value field contains the new data item to be stored. Data-base retrievals and deletions are made by the calls

Get(key,attribute, . . . )
Delete(key,attribute, . . . )

Delete removes a data item and all of its descendants from the data base.

Let us digress momentarily to consider how hierarchical definition structures (compared to the flat structure of Gypsy) are accommodated, while retaining this simple data-base organization. The strategy is to assign a unique name to each unit in a hierarchy (using, for example, consecutive integers) before it is stored in the data base, while maintaining a current context tree and name mapping which together represent the original textual nested structure. This scheme does not affect the methods, but does affect data-base calls in that the key field is replaced by the call:

GetUniqueName(key)

GetUniqueName is a function that, depending on whether or not the "key" is the original name of a program unit or a system-generated name, returns the corresponding unique internal name or "key" itself, respectively.

**Notational Conventions**

In stating parts of the methodology, it is often necessary to specify individual, or sequential instances of the above calls on the data base. To shorten the expression of such calls, the convention adopted is to either use shorthand for the actual call or (when obvious from context) omit a formal designation altogether. The shorthand convention is to subscript an attribute with the key itself.

A few example manipulations on the data-base structure in Fig. 4-8 serve to illustrate the kind of abbreviations used throughout. The phrase "the internal representation of Location_of_max" and the subscripted attribute "$\text{InternalRep}_{\text{Location\_of\_max}}$" both mean

```
Get(Location_of_max,InternalRep)
```

The phrase "set $\text{CheckSemanticsFlag}_{\text{Location\_of\_max}}$" means

```
Insert(Location_of_max,CheckSemanticsFlag,true)
```

whereas a flag is reset by inserting the value false. A phrase such as "add Exchange to the CallsFromInternals attribute of Location_of_max" is equivalent to

```
Insert(Location_of_max,CallsFromInternals,
    cons(Exchange,Get(Location_of_max,CallsFromInternals)))
```

with the Lisp operator "cons" used to add Exchange to the appropriate list.

## 4.3 Defining New Programs and Their Specifications

This section studies the impact of defining new program units on the overall design and verification structure, and vice versa. Generally, newly-defined units are inserted into an existing data-base structure. The methods allow for a unit to be called prior to its definition and for a new unit to call defined or undefined units. The only requirement accompanying this referencing flexibility is that units must be successfully parsed before they are added to the data base.

Key ideas involved in handling new units are illustrated by considering a data base that contains only two program units:

```
function Exchange_sort(A:int_array):int_array =
begin
      .
      .
      .
 Exchange( . . . )
      .
      .
      .
end;


function IsExchanged(A,B:int_array; I,J:int):boolean =
begin
 exit (assume IsExchanged(A,B,i) iff
            (  I in [1..N]  and  J in [1..N]
            and A[I]=B[J] and A[J]=B[I] ) );
end;
```

Observe that both are semantically inconsistent. Exchange_sort has an unresolvable external reference because of its call to Exchange and IsExchanged contains a type mismatch in its exit specification because the recursive call has only three (instead of four) arguments. Suppose that Exchange is now defined and calls IsExchanged. Two questions arise:

- ***Global impact.*** How does Exchange affect Exchange_sort and IsExchanged?

- ***Local impact.*** How do Exchange_sort and IsExchanged affect Exchange?

Regardless of how Exchange is defined, it cannot introduce more inconsistencies in Exchange_sort or IsExchanged. Determining the local impact, however, requires knowing relationships between Exchange and IsExchanged. For example, consider what happens if VCs are generated for the following definition of Exchange:

```
function Exchange(A:int_array; I,J:int):int_array =
begin
  entry I in [1..N] and J in [1..N];
  exit  (all k:[1..N],
           k ne I and k ne J -> Exchange(A,I,J)[k] = A[k] )
    and IsExchanged(A,Exchange(A,I,J),I,J)
    and IsPerm(A,Exchange(A,I,J));
  result := A;
  result[I] := A[J];
  result[J] := A[I];
end;
```

The single VC would be semantically inconsistent because it would contain the inconsistent exit specification of IsExchanged. The methods prevent this kind of occurrence.

The algorithm for defining new programs and specifications uses the following functions. For program unit X, let

InternalCallsTo(X)

return the names of other program units called in the executable body or internal specifications of X and

ExternalSpecCallsTo(X)

return the names of other program units called in the external specifications of X. Also, let the boolean-valued function

SemanticallyConsistent(X)

be a call to a semantic checker to determine whether or not X is semantically consistent. This function is the sole interface between the algorithms and a semantic checker.

The algorithm for adding a new unit consists of four main steps. The first two steps fill in the necessary global cross references. Locally, the third assigns a status value and the fourth determines if any flags need to be set to temporarily override the status setting.

**Algorithm A.** For a newly-defined program unit X, the algorithm is stated as follows:

1. For each Y in InternalCallsTo(X), add X to $\text{CallsFromInternals}_Y$.

2. For each Y in ExternalSpecCallsTo(X), add X to $\text{CallsFromExternalSpecs}_Y$.

3. Set $\text{Status}_X$ to "to be VC generated".

4a. If not SemanticallyConsistent(X), then set $\text{CheckSemanticsFlag}_X$.

4b. Otherwise, if there is a Y in InternalCallsTo(X) or ExternalSpecCallsTo(X) such that $\text{CheckSemanticsFlag}_Y$ or $\text{CheckVCsFlag}_Y$, then set $\text{CheckVCsFlag}_X$.

Step 4b, which is applied when X is semantically consistent, requires further explanation. It decides whether or not a semantic inconsistency can arise in the VCs of X. This can happen, for example, when X calls a unit that is semantically inconsistent, as was just illustrated when Exchange called IsExchanged. Under certain conditions, other inconsistencies in the data-base structure can also arise in the VCs for X. For now, it suffices to say that if any program called by X has CheckSemanticsFlag or CheckVCsFlag set, then $\text{CheckVCsFlag}_X$ is set. It is explained later how this fits into the scheme for preventing things like generating inconsistent VCs for Exchange.

Notice that local cross references, such as CallsFromInternals, are not updated. Perhaps an additional step is needed! Actually, calls to undefined units are filled in prior to their definition by the first two steps above or by the algorithm in Section 4.4. When Exchange_sort called Exchange in the previous example, this fact was noted immediately in the data base before Exchange was defined.

## 4.4 Revising Programs and Their Specifications

A difficult problem in incremental environments is how to effectively determine the effects of changes to existing data. Section 4-1 outlined a general approach to this problem which is formalized here. Sections 4.5 and 4.6 present some ideas (not previewed in Section 4.1) for performing a finer analysis of specifications and VCs, respectively. Some auxiliary functions are introduced before the main algorithm for handling revisions is stated.

```
1    function Location_of_max(A:int_array; I,J:int):int =
2    begin
3      entry I in [1..N] and J in [1..N] and I le J;
4      exit Location_of_max(A,I,J) in [I..J]
5        and A[Location_of_max(A,I,J)] = Value_of_max(A,I,J);
6      var K:int := I;
7      keep I le J and I in [1..N] and J in [1..N] and K in [1..N];
8      result := I;
9      loop
10       assert A[result] = Value_of_max(A,I,K)
11         and result in [I..K] and K in [I..J];
12       if K ge J then leave end;
13       K := K+1;
14       if A[result] < A[k] then result := k end;
15     end;
16   end;
```

Figure 4-10. Definition of Location_of_max

Four boolean-valued functions are used for determining what parts of a unit X are changed in X'. Examples of different parts considered are taken from the definition of Location_of_max in Fig. 4-10. The first needed function is

ChangeInHeader(X,X')

which indicates whether the header of X has been revised in X'. The header for Location_of_max is on line 1. The function

ChangeInExternalSpecs(X,X')

detects changes in external specifications (lines 3-5). These are the only two functions needed to detect changes that have global effects. Next, changes to external or internal specifications are detected by

ChangeInSpecs(X,X')

Changes to these specifications can have local effects on the assocated unit. Changes to lines 3, 4, 5, 7, 10, and 11 can affect the verification of Location_of_max. The function

ChangeInBody(X,X')

detects changes to lines 2-16, excluding specifications.

All four of these functions detect semantic as well as syntactic differences. For example, ChangeInExternalSpecs checks for exit specifications that are syntactically identical, but have variables of different types. Free variables may be bound to one type in the parameter list of X and to another in X'.

The two functions given below are used by several algorithms for verification purposes. They both reflect the policy of fully expanding VCs, i.e., of assuming that all potentially-usable specifications are in fact used in proofs. Section 4-10 changes these functions to account for those specifications actually used.

The function MarkToCheckVCs traces the effects of an inconsistent external specification through the data-base structure, marking those units whose verification is affected. For a unit X,

```
MarkToCheckVCs(X)=
      for each Y in CallsFromExternalSpecs_X or CallsFromInternals_X
            (set CheckVCsFlag_Y;
             if Y in CallsFromExternalSpecs_X then
                  MarkToCheckVCs(Y) )
```

Suppose that X has a type conflict in its exit specification. Then, MarkToCheckVCs sets CheckVCsFlag for those units that have the exit specification of X in at least one of their VCs. These affected units are identified by tracing upward through the graph of calling relationships, beginning with X. Traversal of a path terminates when the inconsistent specification can be exported no further -- i.e., when it cannot be exported to another unit via externally-visible specifications. For units not having VCs, the algorithms work independent of whether or not MarkToCheckVCs sets their CheckVCsFlag.

The function IsPathConsistent determines whether there is a semantic inconsistency in the data base that would affect at least one of the VCs for unit X.

```
IsPathConsistent(X)=
        if CheckSemanticsFlag_X then false
        else if CheckVCsFlag_X then
            (if IsPathConsistent(Y) is true for every Y in ExternalSpecCallsTo(X)
                then true
             else false )
        else true ;
```

IsPathConsistent is, in a sense, the inverse of MarkToCheckVCs. MarkToCheckVCs traversed upward paths to find all units to which potentially inconsistent specifications could be exported, whereas IsPathConsistent traverses downward paths to see if there is a potentially inconsistent specification that would be imported. Again, the key observation is that inconsistencies propagate only through externally-visible specifications.

A five-step algorithm for handling changes to a unit follows. The first four steps perform the necessary global analysis, marking, and updating, while the fifth step handles local tasks. The entire algorithm is formally stated, then each main step is explained.

**Algorithm R.** If a program unit X is replaced by the unit X', then the following steps are performed in order.

1. ***Check the header.*** If ChangeInHeader(X,X'), set $\text{CheckSemanticsFlag}_Y$ for each Y in $\text{CallsFromInternals}_{X'}$ or $\text{CallsFromExternalSpecs}_{X'}$, execute MarkToCheckVCs(Y) for each Y in $\text{CallsFromExternalSpecs}_{X'}$, and go to step 3.

2. ***Check external specifications.*** If ChangeInExternalSpecs(X,X'), then (if SemanticallyConsistent(X), SemanticallyConsistent(X'), and IsPathConsistent(Y) is true for each Y in ExternalSpecCallsTo(X) and ExternalSpecCallsTo(X') then (call MarkToCheckVCs(X') if Algorithm C is false) else call MarkToCheckVCs(X')).

3. ***Delete cross references.*** For each Y in InternalCallsTo(X) and not in InternalCallsTo(X'), delete X from $\text{CallsFromInternals}_Y$. Perform this step again replacing InternalCallsTo by ExternalSpecCallsTo and CallsFromInternals by CallsFromExternalSpecs.

4. ***Add cross references.*** For each Y in InternalCallsTo(X') and not in InternalCallsTo(X), add X' to $\text{CallsFromInternals}_Y$. Perform this step again replacing InternalCallsTo by ExternalSpecCallsTo and CallsFromInternals by CallsFromExternalSpecs.

5. ***Local updating.*** Perform the following steps in order until one succeeds.

5a. ***Check semantics.*** If not SemanticallyConsistent(X'), then set $CheckSemanticsFlag_{X'}$.

5b. ***Prevent import of inconsistent specifications.*** If there is a Y in ExternalSpecCallsTo(X') or InternalCallsTo(X') such that IsPathConsistent(Y) is false, then set $CheckVCsFlag_{X'}$.

5c. ***Any previous proofs?*** If $Status_X$ is "no proofs established", then set $Status_{X'}$ to "to be VC generated".

5d. ***Keep intact all proofs.*** If not ChangeInBody (X,X'), SemanticallyConsistent(X), $CheckVCsFlag_X$ is not set, ChangeInSpecs(X,X'), and Algorithm L is true, then set $Status_{X'}$ to $Status_X$.

5e. ***Must check proofs individually.*** Set $CheckVCsFlag_{X'}$.

Algorithms G and L determine the global and local impact of changes to specifications (see Section 4.5). Omitted, for clarity, from this and other algorithms are explicit statements of "inherited" attribute:value pairs. The reader should assume that such inheritance always occurs unless stated otherwise. In step 1, for example, $CallsFromInternals_{X'}$ is the same as $CallsFromInternals_X$.

Figure 4-11 presents another data-base structure that will be used for expository purposes. As with the sorting example, names represent program units (basis properties will be added later) and arrows indicate calls.

Step 1 determines the global impact of changing the header of a program. Suppose that X takes two arguments and is replaced by X' which takes only one. Step 1 ensures that all direct callers of X' (viz., A and B) are checked for semantic consistency. After A and B are marked, MarkToCheckVCs marks all units that have at least one VC containing a type mismatch. If, for example, the external specifications of A call X' and those of D call A, all markings performed by this step are illustrated in Fig. 4-12. Notice that H, F, and G are not affected in any way by the change to X. H is not affected because the external specifications of C do not call A, meaning that any inconsistent specification in C cannot be exported to H. Similarly, F and G are not affected by the change, because the external specifications of B do not call X'.

Step 2 determines the global effects of changing the external specifications of X. Algorithm G is called to determine if the needed logical relationship holds between the external specifications of X and X'. Algorithm G does not apply when these specifications either

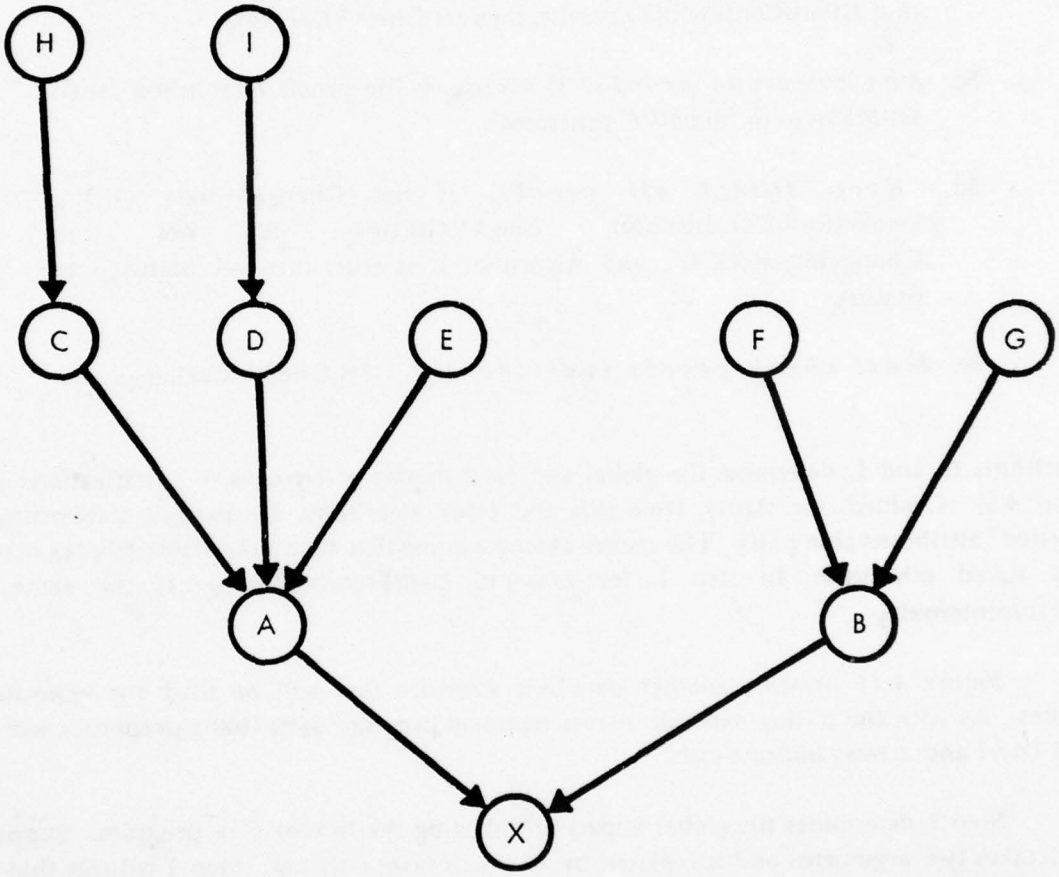



Figure 4-11. Another example of a data-base structure

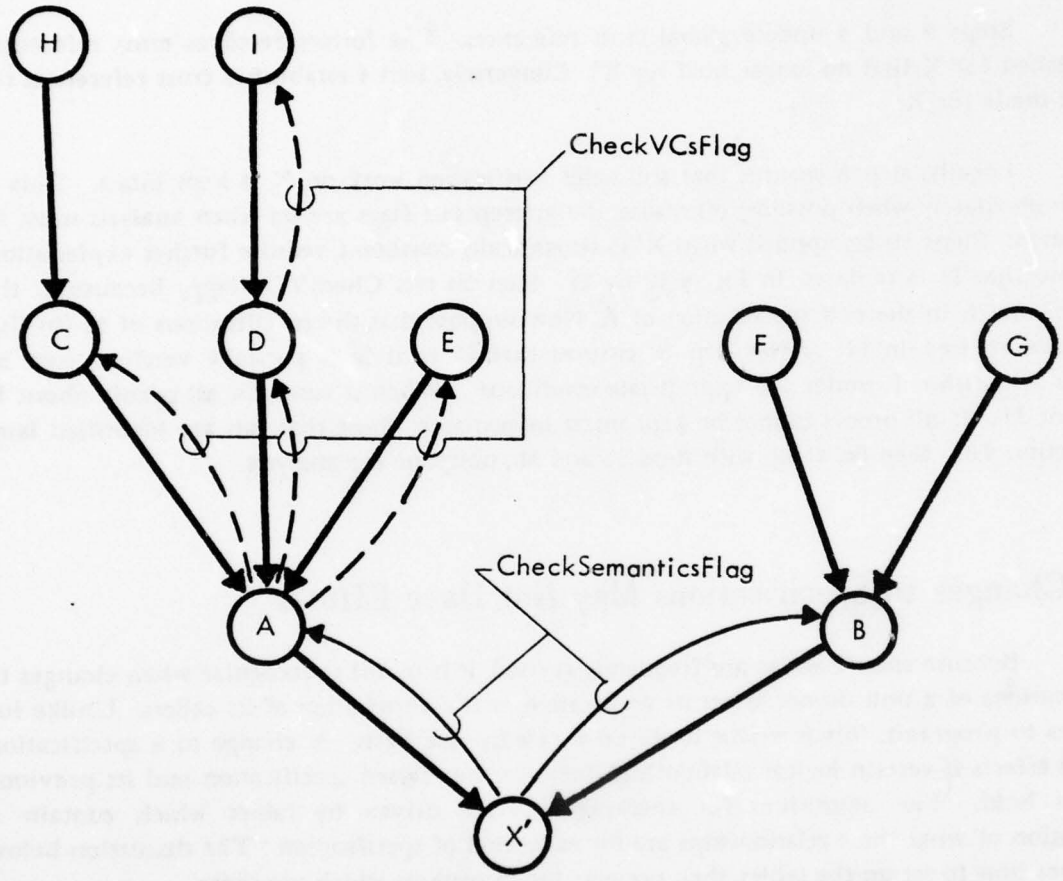



Figure 4-12. Potential effects of revising the header of X

contain type conflicts or can have inconsistencies imported into the proofs. This latter condition again reflects the conservative view of fully expanding specifications. When Algorithm G fails or cannot be called, MarkToCheckVCs marks the same units as does step 1. The only difference being that direct callers are marked as being verification (instead of design) inconsistent.

Steps 3 and 4 update global cross references. The former removes cross references established for X that no longer hold for X'. Conversely, step 4 establishes cross references for X' not made for X.

Locally, step 5 ensures that still-valid verification work on X is kept intact. This is done immediately when possible; otherwise, the appropriate flags are set when analysis must be postponed. Steps 5b-5e, applied when X' is semantically consistent, require further explanation. Suppose that D is replaced in Fig. 4-12 by D'. Step 5b sets $\text{CheckVCsFlag}_{D'}$ because of the type mismatch in the exit specification of A. Now suppose that the specifications of H (in Fig. 4-12) are revised in H'. After step 5c ensures that H is at least partially verified, step 5d applies Algorithm L under the appropriate conditions. When it succeeds, all proofs about H hold for H'. If all proofs cannot be kept intact immediately, those that can are identified later (cf. Section 4.6). Step 5e, along with steps 5a and 5b, postpone the analysis.

## 4.5 Changes to Specifications May Not Have Effects

Because specifications are frequently revised, it is useful to recognize when changes to specifications of a unit do not affect its verification or the verification of its callers. Unlike for changes to programs, this is viable to do on a case-by-case basis. A change to a specification has no effects if certain logical relationships between the revised specification and its previous version hold. The algorithms for checking this are driven by tables which contain a description of what these relationships are for each kind of specification. The discussion below describes how to set up the tables, then presents the algorithms which use them.

### Setting Up The Tables

Table entries for specifications consist of three parts:

- The textual name used to identify the specification.
- A designation of a formula relating a revised specification to its previous version. These formulas, of course, vary according to the program proof techniques employed. If omitted, a default of false is assigned.

- An indication as to whether failure to prove the designated formula has local or global effects. Changes to some specifications can have both kinds of effects, in which case two table entries are required. Traditional entry and exit specifications are in this category.

Of course, it does not matter whether local or global descriptions are actually stored in separate tables or combined into one.

As a specific example of how to describe specifications, we will develop a set of local and global table entries for entry and exit specifications, beginning with the local entries. Table declarations are of the form

```
Declare(effect
        (specification-name     formula)
                ...
        (specification-name     formula))
```

Using this template, a description for determining local effects is

```
Declare (local
        (entry     entry'⊃entry)
        (exit       exit⊃exit'))
```

This states that changes to entry specifications do not affect the verification of the associated program if the changed entry specification implies the unchanged one. The converse relation must hold for exit specifications. Both descriptions follow from Hoare's rule of consequence. †

Globally, entry and exit specifications of a program can arise in the VCs of its callers in several ways. Suppose that the proof strategy handles program calls by proving that entry specifications hold at all calling sites and by adding formulas of the form entry⊃exit (composed of specifications from callees) to VCs of callers. †† Then, the description for global effects is

---

† The rule of consequence is:

$$\frac{P \supset Q,\ Q\{S\}R,\ R \supset T}{P\{S\}T}$$

†† SID employs this scheme, among others. It instantiates and proves entry specifications at all calling sites and adds uninstantiated formulas of the form entry⊃exit to VCs during proofs.

```
Declare (global
        (entry    entry⊃entry')
        (exit     if entry⊃entry' then exit'⊃exit
                  else (entry'⊃exit') ⊃ (entry⊃exit)))
```

This says that changing an entry specification has no global effects when the unchanged entry emplies the changed entry. The exit relationship is described as an if-then-else statement to take advantage of knowledge about what has changed. If the entry has not changed since the last time the exit changed, the test is automatically true and a proof of the simpler formula on the then branch is attempted.

SID uses this kind of knowledge, as is evident in the following interchange.

```
<- What does changing its exit assertion affect?

     "its" refers to Location_of_max.

      .
      .
      .

The verification of other programs is not affected if the formula
   Changed exit specification
 -->
  C1: LOCATION_OF_MAX(A, I, J) in [I..J]
  C2: A[LOCATION_OF_MAX(A, I, J)] = VALUE_OF_MAX(A, I, J)
is true. If not, the change invalidates the verification of EXCHANGE_SORT.
```

The formulas in the answer come from the above if-then-else rule for exit specifications. Since the entry specification of Location_of_max has not changed, the answer refers only to its exit specifications. When SID actually attempts to prove such formulas, all free variables are bound and typed. The type information comes from the symbol tables.

Global descriptions follow from the following sound, first-order inference rule:

$$\frac{\forall \ldots (H_1 \supset H_2), \forall \ldots (H_2 \wedge H_3 \supset C)}{\forall \ldots (H_1 \wedge H_3 \supset C)}$$

$H_1$, $H_2$, $H_3$, and C are first-order formulas whose free variables are universally quantified as indicated above. This rule is called the *rule of revision*. Figure 4-13 illustrates its use in deriving the global entry and exit descriptions given above. Only the then branch of the if-

then-else formula is derived; the else branch derivation is identical, except for the obvious substitutions.

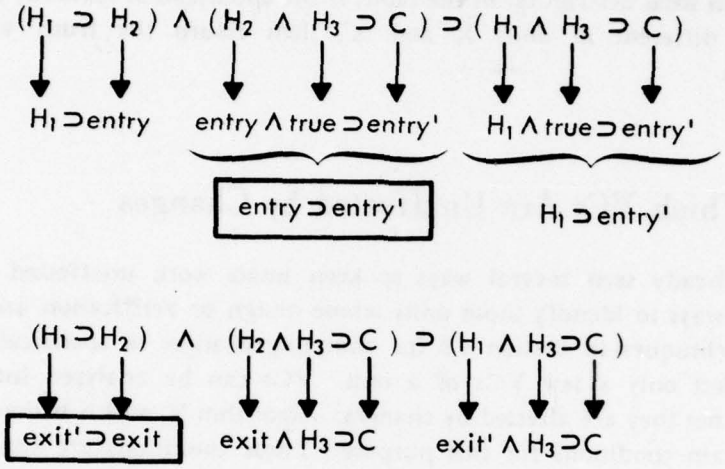



Figure 4-13. Deriving entry and exit descriptions from the rule of revision

Other specifications in Gypsy are described similarly. Those discussed later are basis properties in Section 4.8 and data abstraction specifications in Section 4.9.

### Algorithms for Manipulating the Tables

It is now trivial to write the algorithms for analyzing changes to specifications. Two are called from Algorithm R. The first is Algorithm G, which determines if changes to external specifications have global effects. Algorithm L determines if changes to internal or external specifications have local effects on the associated unit.

**Algorithm G.** For each global description in the table, if the specification textually identified by specification-name is different in units X and X', then return the truth value of the corresponding formula.

Before executing this algorithm, SID first sees if there is at least one unit traversed by

MarkToCheckVCs(X') that is partially or completely verified. Effort expended in trying to prove the designated formulas is wasted unless succeeding proofs will keep intact previously-done verification work.

**Algorithm L.** For each local description in the table, if the specification textually identified by specification-name is different in units X and X', then return the truth value of the corresponding formula.

## 4.6 Deducing Which VCs Are Unaffected by Changes

We have already seen several ways to keep intact work unaffected by changes. Section 4.4 described ways to identify those units whose design or verification are affected by changes, using the techniques in Section 4.5 for analyzing changes to specifications. Often, however, changes affect only a few VCs of a unit. VCs can be analyzed individually to determine whether or not they are affected by changes. Algorithm V, which is described below, is applied under certain conditions for this purpose. These conditions are enforced by the algorithm in Section 4.7 for resolving inconsistencies.

The algorithm works by "comparing" new and old VCs to determine if their proofs coincide. Whatever the scheme for doing this comparison, both syntactic and semantic concerns must be satisfied. Algorithm V uses a simple "equivalence" test -- the new and old VC must be the same syntactically within a uniform change of variables and semantically symbol for symbol. Uniform variable renaming is added as a special case so that the common source program change of uniformly renaming an identifier will not necessitate any reverifying.

**Algorithm V.** Let VC designate a verification condition of X and VC' a verification condition of X'. For every VC',

1. If there is an at least partially proved verification condition VC such that it and VC' satisfy the equivalence test described above, then set the status of VC' to that of VC.

2. Otherwise, the status of VC' becomes "to be proved".

When checking the semantic criteria for equivalence, the type of free variables in VC' are in $SymTab_{X'}$ and those in VC are in $SymTab_{X}$. In SID, this algorithm is not applied to VCs that simplify to true during generation.

Aware that type errors in X may only affect some of its VCs and their proofs, Algorithm V keeps intact those parts that hold for X'. The equivalence test always fails for

those VCs of X that contain type errors, because VCs for X' are type correct (cf. Section 4.7). For example, a verification condition with the wrong number of arguments to a function causes the syntactic comparison to fail. Things like a wrong argument type in a function call or a wrong variable type in an expression cause the semantic comparison to fail.

There are many variations on Algorithm V and several practical tradeoffs among them. One is showing that previously-proved VCs are semantically consistent and that they imply new ones, establishing the latter by modus ponens. Another is attempting recovery at the subgoal level, rather than the VC level -- implying several complexities mostly related to theorem proving. These and some other variations require more processing (in varying degrees) than Algorithm V and may also require human assistance. Algorithm V is straightforward and requires no human intervention.

## 4.7 Resolving Inconsistencies

This section describes the policy for determining which manipulations can be performed on data-base structures. The policy does not consider operations that do not change the state of the data base (e.g., display operations) because they can always be executed. It also omits from consideration the always-allowed operation of translating data. Placing no restrictions on when translation can occur enables the user to add to or revise the data base at his convenience.

The policy adopted must embody some definition of what it means for an operation to be *legitimate*. Here, an operation is legitimate if it makes sense and if it would cause a consistent state change on the data base. A simple status mechanism suffices to limit the applicable operations when everything in the data base is consistent. For example, if the status of a unit is "to be VC generated", a "prove VCs" command is nonsensical. Complexities arise here because the data base frequently contains inconsistent data, requiring that the local state of development and the global data-base configuration be taken into account.

The following algorithm views the semantic checking of a unit as an operation that cannot be requested -- it is performed automatically by the algorithm when needed. Reformulating the steps without this assumption is trivial. Steps 1 - 3 ensure that only consistent state-changing operations are performed; step 4 ensures that the operation makes sense in situations where inconsistencies are not possible.

**Algorithm R1.** For a unit X, perform the following steps in order:

1. If $CheckSemanticsFlag_X$ or $CheckVCsFlag_X$ is set, then if IsPathConsistent (Y) is false for some Y in InternalCallsTo (X) or ExternalSpecCallsTo (X), the operation is not allowed.

2. If $CheckSemanticsFlag_X$ is set, then

   2a. If SemanticallyConsistent(X) and the operation is "generate VCs", then generate VCs for X, execute Algorithm V, reset $CheckSemanticsFlag_X$ and $CheckVCsFlag_X$, and update $Status_X$.

   2b. Otherwise, the operation is not allowed.

3. If $CheckVCsFlag_X$ is set, then

   3a. If the operation is "generate VCs", then generate VCs for X, execute Algorithm V, reset $CheckVCsFlag_X$, and update $Status_X$.

   3b. Otherwise, the operation is not allowed.

4. If $Status_X$ indicates that the operation is legitimate, then perform it and update $Status_X$, otherwise the operation is not allowed.

Step 1 prevents inconsistencies in the data base from causing an inconsistent state change. In Fig. 4-12, this step ensures that the VCs for I, C, D, and E are not generated until the semantic inconsistency is removed from the external specification of A.

Step 2 resets CheckSemanticsFlag only when a unit is semantically consistent and the operation is "generate VCs". If VCs are generated, Algorithm V attempts to keep intact previous proofs that remain valid. Then the new state of development is reflected in the data base by resetting the flags and updating the status for the associated unit. Further, it prevents the VCs for A and B from being generated until A and B are semantically inconsistent with X'.

Updating the status of a unit is a straightforward process performed only when it has no inconsistencies. The exact procedure, however, depends upon the characteristics of the underlying design and verification system. For example, if the status of a unit is "to be VC generated", after VC generation the status may be one of "to be proved", "partially proved", "proved", etc. The actual setting depends on whether the VC generator combines proving with the VC-generation process.

Step 3 handles units whose only set flag is CheckVCsFlag. It does not apply, for example, to units A and B of Fig. 4-12. CheckVCsFlag is reset after generating VCs for the current unit and applying Algorithm V. The status is updated as before.

Step 4 is applicable only when the current unit is design and verification consistent. $Status_X$ delimits the set of legitimate operations and is updated when an operation is performed. For example, if $Status_X$ is "to be VC generated", then the "prove VCs" operation is not legitimate.

Let us now see how Algorithm R1 applies to Fig. 4-12. Units H, F, and G can always be manipulated by step 4 because they do not contain and cannot be affected by inconsistencies in the structure; X' can be operated on by step 2. Operations on all other units are ruled out by step 1, since a state change cannot be guaranteed to be consistent. Inconsistencies in X' and then A must be removed by step 2a before C, D, E, and I can have VCs generated by step 3a.

## 4.8 Basis Properties: Lemmas, Rewrite Rules, and Definitions

We are now ready to add basis properties to the data-base structure. These properties typically call functions already in the data base. Figure 4-14 illustrates how such calls are reflected in the data base by adding basis properties $B_1$, $B_2$, and $B_3$ to Fig. 4-12. X, I, and F are functions called by $B_1$, $B_2$, and $B_3$, respectively. Using basis properties in proofs ties them into the data-base structure even more intimately.

In this extended data base, changes to basis properties can affect verifications, and changes to programs can affect basis properties and the proofs in which they were used. Both situations are studied in this section. Included in the discussion are specific rules for showing that changes to basis properties have no effects. These rules, like the ones for specifications, describe needed logical relationships between a revised basis property and its previous version.

The different kinds of basis properties discussed (which are the kinds handled in SID) are lemmas, rewrite rules, conditional rewrite rules, and definitions. All are assumed to be syntactically and semantically checkable. As a result, things like function calls in lemmas must be checked for the right number and type of arguments. Of course, the needed information may be either user-defined (such as function definitions in the data base) or built into the semantic checker.

### Changes to Basis Properties May Not Have Effects

For a new or changed basis property B, the following steps perform the necessary local analysis:

1. If not SemanticallyConsistent(B), set $CheckSemanticsFlag_B$.

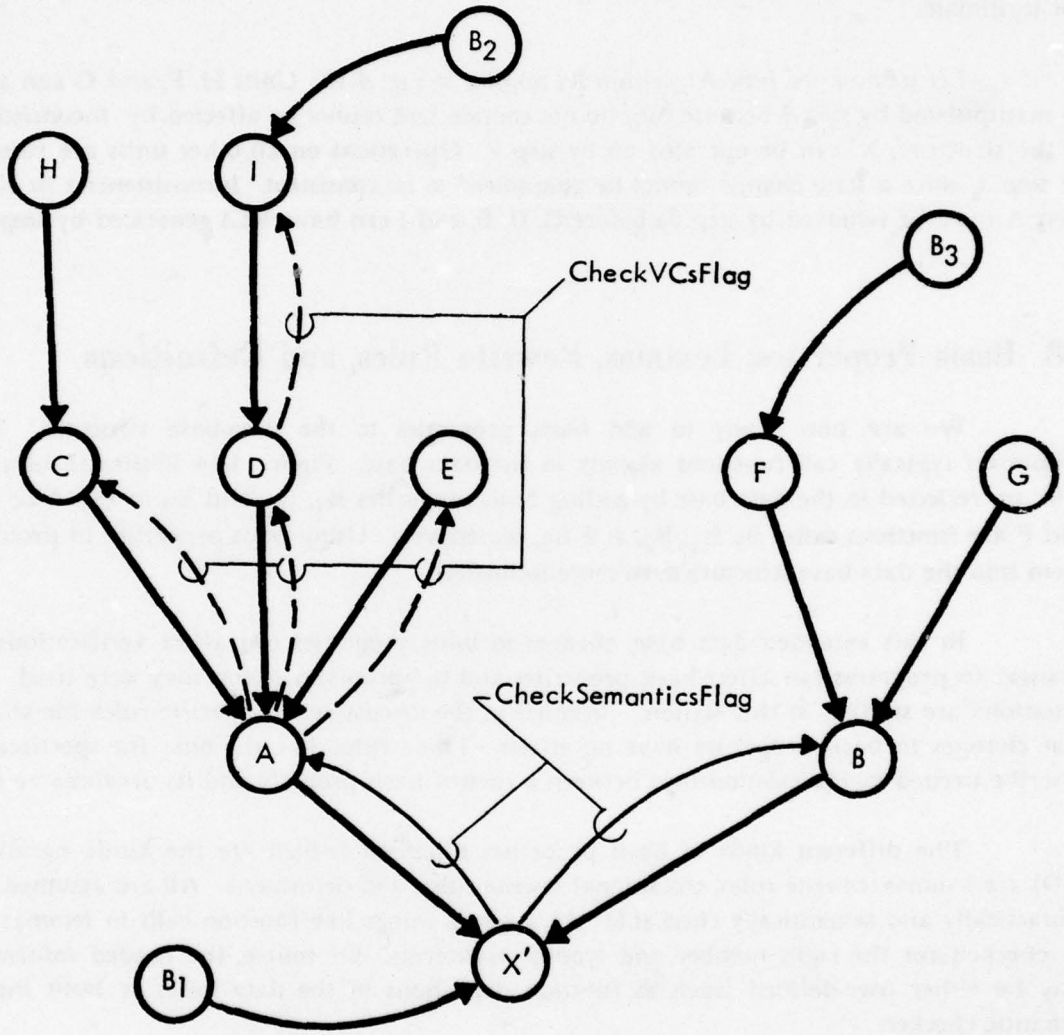



Figure 4-14. Adding basis properties to Fig. 4-12

2. Otherwise, if the data base contains a function X called by B such that IsPathConsistent(X) is false, then set $NotUsableFlag_B$.

Step 1 ensures that a semantically inconsistent basis property is not used in proofs; step 2 keeps semantically inconsistent specifications from being added to proofs through basis properties. The setting of $NotUsableFlag_B$ or $CheckSemanticsFlag_B$ prohibits use of B in proofs until both are reset. $NotUsableFlag_B$ is reset when inconsistent specifications can no longer be introduced in proofs via B. This is determined analogously to step 1 of Algorithm RI. $CheckSemanticsFlag_B$ is reset when all semantic errors are removed from B.

Changes to basis properties also can have an impact on the rest of the data base. The procedure for determining this global impact is highlighted below.

1. If not (SemanticallyConsistent(B) and SemanticallyConsistent(B')), or if IsPathConsistent(Y) is false for at least one function Y called by B or B', or if the comparison described below between B and B' is not successful, then add B to $CheckBasisFlag_X$ for each X in $UsedIn_B$.

2. Otherwise, no proofs are affected.

The conservative set of conditions given in step 1 are the same as those required to compare two specifications. Comparison rules for the four kinds of basis properties are given below.

***Lemmas.*** For a lemma L and its replacement L', the rule of revision says that if the formula

$$L' \supset L$$

is true, then proofs depending on L remain valid for L'.

What happens if a basis property is deleted, rather than revised? If the deleted property is shown to be logically valid, proofs depending on it remain valid. That is, they can be done without it.

***Rewrite rules.*** A rewrite rule is represented as

$$A \dashrightarrow B$$

and is applied by changing all subformulas of the form A into the form B, but not vice versa. The underlying semantics of these rules in SID is equality. That is, if a theorem is proved using A --> B, it can also be proved using A = B. This observation together with the rule of revision indicates that establishing the formula

$$(A' = B') \supset (A = B)$$

shows that proofs using the rewrite rule A --> B can also be done using A' -- > B'.

***Conditional rewrite rules.*** These are slight generalizations of rewrite rules in which a rewrite is performed only if a given condition is true. Proofs using a rewrite rule A --> B when condition C is true remain valid for its replacement whenever the formula

$$(C' \supset A' = B') \supset (C \supset A = B)$$

is proved.

***Definitions.*** A definition is a statement that establishes the meaning of an expression. An example of how they arise in Gypsy is the definition of IsExchanged on lines 68-70 of Fig. 4-1. Generally, the definition for an n-place function F can be stated as

$$F = E$$

F is the definiendum and the definien E is an expression. Definitions behave like rewrite rules (replacing F by E) in proofs (as illustrated by the expansion of the definition of IsExchanged in Section 3-3). Showing that

$$E' = E$$

holds when the definien changes indicates that proofs depending on E remain valid for E'. Changes to F are handled by Algorithms R and V.

### Changes to Programs May Affect Basis Properties and Proofs

Adding basis properties requires minor extensions to Algorithms R and V. Algorithm R must account for the fact that a change to the header of a function may cause a type conflict in all calling basis properties, in turn rendering proofs in which they were used inconsistent. Step 1 of Algorithm R, using CallsFromBasisProps to identify calling properties, is easily extended for this purpose.

Another extension is illustrated with the aid of Fig. 4-14. With the addition of basis properties, MarkToCheckVCs must alter its behavior. Whenever it encounters a program unit that can export an inconsistency, it must set NotUsableFlag for each calling basis property. Setting NotUsableFlag in turn causes CheckBasisFlag to be set for each unit that used the property in its proof. Figure 4-15 illustrates the complete set of markings for Fig. 4-13, if $B_1$ was used in the proof of C and if I can export an inconsistency to its callers.

Algorithm V must exclude VCs whose proofs use a subsequently invalidated basis

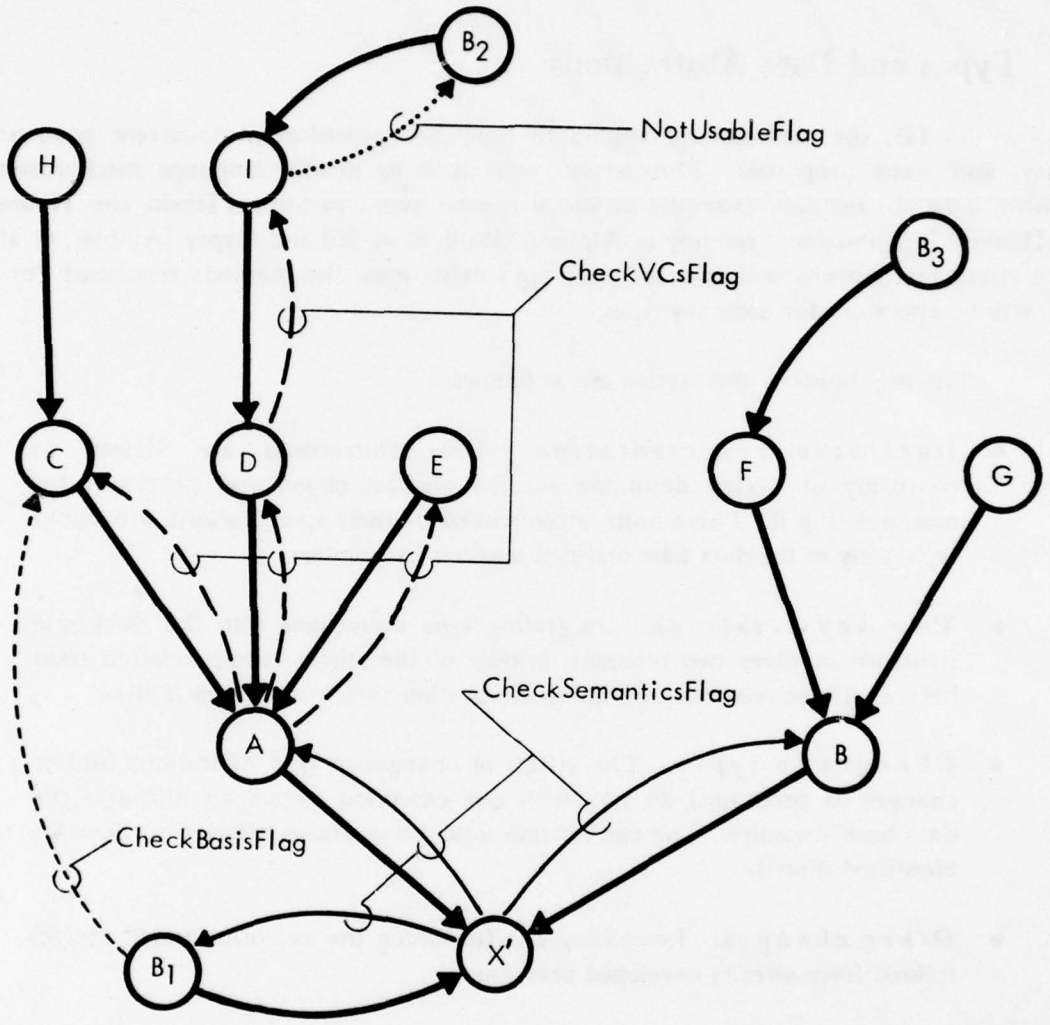



Figure 4-15. Changes to the header of X can affect basis properties and proofs

property from consideration. A VC is excluded if the intersection of the basis property names in CheckBasisFlag with those in the VC's Used attribute is non-empty.

## 4.9 Types and Data Abstractions

So far, the methodology applies to functions, procedures, concurrent processes of Gypsy, and basis properties. This section extends it to handle language mechanisms for defining data abstractions, examples of which can be found in Simula [Dahl and Hoare 72], Clu [Liskov 75], and more recently in Alphard [Wulf, et al. 76] and Gypsy [Ambler, et al. 77]. Since these mechanisms subsume ordinary type definitions, the methods described for data abstractions also work for ordinary types.

The highlights of this section are as follows:

- ***Data-base representation.*** Data abstractions are viewed as consisting of a type definition for the abstract object and programs for manipulating it. These units, accompanied by their specifications, are stored separately in the data base and tied together by pointers.

- ***Two key relations.*** Integrating type definitions into the data-base structure involves two relations, instead of the single calling relation used before. These relations distinguish declaration sites from reference sites.

- ***Changes to types.*** The effects of changes to type definitions (unlike changes to programs) do not, with one exception, propagate through the data-base structure. The two-relation description allows affected units to be identified directly.

- ***Other changes.*** Everything else (including the exception noted above) follows from already-developed principles.

The following outlines the main details surrounding each of these points, beginning with a discussion of extensions to the data base.

### Representing Data Abstractions in the Data Base

A data abstraction is viewed as consisting of a restricted type definition for the abstract object and operations that manipulate the internal structure of the object. The word "restricted" is used to emphasize that users of the abstraction do not have access to its internal

structure -- only the associated operations do. The representation given below suffices for describing data abstractions written in Gypsy. Of course, additions or deletions may be needed for other languages.

Restricted type definitions may have four parts:

1. A header that includes the name of the abstract object and an optional parameter list.

2. External specifications about parameters. These specifications must hold whenever an instance of the type is declared.

3. External specifications that can be assumed in proofs of all users of the abstraction (e.g., abstract axioms).

4. A body which defines the implementation of the abstract object.

Declarations that create instances of an abstract object refer to the entities in part 1 and must also satisfy any restrictions on parameters in part 2. The specifications in part 3, in contrast to those in part 2, hold everywhere the abstraction is referenced, except possibly for states within units that access the internal structure of the restricted type. The body of a restricted type unit is visible only to those units given access.

Program units that access restricted types are just like other program units, except for some additional specifications. These specifications are used in showing the needed abstract-concrete relationship, i.e., that the implementation "models" the abstraction.

Four new attribute:value pairs are added to the data base. The first three are associated only with restricted type units; the last only with units that have access to the body of a restricted type.

| | |
|---|---|
| **AbstractOps** | is a list of the units given access to the body of the restricted type. |
| **DeclaredIn** | is a list of the units (including other type definitions) in which an instance of the restricted type is declared. |
| **ReferencesFrom** | is a list of the units that reference the abstract object, including units in DeclaredIn. |
| **CheckMappingFlag** | is a boolean flag that indicates whether the abstract-concrete relation must be checked. |

CheckSemanticsFlag and CheckVCsFlag are used for both kinds of units and have the same meaning as before. CheckSemanticsFlag still has the highest flag precedence, subsuming both CheckVCsFlag and CheckMappingFlag.

If the unit of interest is an ordinary type definition, this scenario still applies. A type unit is a restricted type unit having only parts 1 and 4 and with AbstractOps set to the union of all units that are in DeclaredIn and ReferencesFrom, since all those units can access its body.

**Changes to Restricted Types**

The first five steps of the following algorithm are concerned with determining the global effects of changing restricted type definitions and the sixth determines the local effects.

**Algorithm T.** If a restricted type T is replaced by T', the following steps are performed.

1. If *part 1* is changed, then set CheckSemanticsFlag$_X$ for each X in DeclaredIn$_T$.

2. If not SemanticallyConsistent(T'), set CheckVCsFlag$_X$ for each X in ReferencesFrom$_T$ and set CheckSemanticsFlag$_Y$ for each Y in AbstractOps$_{T'}$, and go to step 6.

3. If *part 2* is changed and IsPathConsistent(Y) is true for each Y called in part 2 of T and T', then set CheckVCsFlag$_X$ for each X in DeclaredIn$_T$ if Algorithm DAG is false.

4. If *part 3* is changed and IsPathConsistent(Y) is true for each Y called in part 3 of T and T', then set CheckVCsFlag$_X$ for each X in ReferencesFrom$_T$ if Algorithm DAG is false.

5. If *part 4* is changed, set CheckSemanticsFlag$_X$ for each X in AbstractOps$_{T'}$.

6a. If not SemanticallyConsistent(T'), set CheckSemanticsFlag$_{T'}$.

6b. Otherwise, if CheckVCsFlag$_T$ is not set and *part 2* or *3* has changed and Algorithm DAL is false, then set CheckVCsFlag$_{T'}$.

Algorithms DAG and DAL determine the impact of changes to specifications, just like Algorithms G and L in Section 4.5.

Step 1 ensures that all declaration sites are checked for semantic errors when the

header of T is revised in T'. If, for example, T' takes a different number of arguments than T, all places where T was declared now have a type mismatch.

Step 2 determines the effects of replacing T with a semantically inconsistent T'. Suppose that an abstract axiom of T is not only changed in T', but also contains a type error. This error prevents the two versions of the axioms from being logically compared. Therefore, all proofs using the axiom from T are currently invalid. This is reflected in the data-base structure by setting CheckVCsFlag for every unit that declares or references an instance of T.

Step 2 also sets CheckSemanticsFlag for each unit that has access to the internal structure of T. A semantic inconsistency in T' may in turn cause all references to its body to be inconsistent. If this step is executed, steps 3-5 are skipped.

Steps 3-5 deal with changes to other parts of restricted types. Steps 3-4 isolate VCs affected by changes to external specifications. If the body of a type is revised, step 5 sees that all units accessing the internal structure of the type are checked for conflicts.

Locally, step 6 keeps instact proofs done for T that hold for T'.

***One exception.*** It was mentioned at the beginning of this section that there is one situation in which the effects of changes to types propagate like changes to programs. Consider the following example:

```
type  type1 = record(field1: array ((1..10)) of integer;
                     field2: array ((1..10)) of integer);

function f( ) =
begin
exit  some x:type1, . . . x.field1 . . . ;
  .
  .
  .
end;
```

type1 is a record with two fields, an instance of which is declared in the exit specification of f. Suppose that the representation of type1 is now changed to

```
type   type1 = array ((1..20)) of integer;
```

Notice the effect this change has on the exit specification of f. It now contains a type error because the record notation "x.field1" is no longer appropriate. Even though callers of f may not reference type1, their verification is invalidated if they imported the exit specification of f for use in proofs.

Similarly, inconsistent data type axioms can invalidate proofs of units that do not reference the type. In the above example, the axioms for type1 can be used everywhere the exit specification of f is used.

### Changes to Units That Access Restricted Types

These units differ from ordinary functions and procedures in one significant way -- some of their specifications are used to establish the needed abstract-concrete relationship. Algorithm R, due to the way data abstractions are represented in the data base, applies almost directly. The main revision is that the local part of Algorithm R (step 5) must set CheckMappingFlag whenever the abstract-concrete relation needs to be reestablished.

To resolve inconsistencies, Algorithm RI must know when to reset CheckMappingFlag. After ensuring that a restricted type is semantically consistent, step 2a resets it. Also, an additional step, either immediately before or after step 3, must be added to reset CheckMappingFlag when the operation "generate abstract-concrete VCs" is performed.

### Changes to Other Kinds of Units

Data abstractions must be taken into account when other kinds of units change. This mainly affects the function MarkToCheckVCs, introduced in Section 4.4 for propagating the effects of changes through the data-base structure. Wherever it encounters a restricted type unit, it must perform the same markings as Algorithm T for each affected part. MarkToCheckVCs must also recognize units that access restricted types and know when to set their CheckMappingFlag. Introducing a data-base entry called CallsFromMappingSpecs, analagous to CallsFromExternalSpecs and CallsFromInternalSpecs, makes this recognition trivial.

## 4.10 Implementation Notes

For use in SID, the formalization presented in this chapter was altered to:

- Invalidate only those proofs that are *actually* (not just potentially) affected by changes to specifications.
- Allow for entry specifications to be proved at all calling sites.

Actually, more than just an alteration to the methods is needed to invalidate only affected proofs. Recall that this chapter assumed that specifications from called programs are fully expanded during VC generation. And when a change to a specification has effects, all VCs containing that specification are invalidated.

But what if the specification was not actually used in some of the proofs? SID does not invalidate them. Recall from Section 3.3 how SID adds specifications to VCs. During VC generation, only references to specifications of called programs (instead of the specifications themselves) are added to VCs. Then, complete specifications, or parts of specifications, that do not contain type errors are expanded as needed during proofs. Every expansion is recorded in the data base. The altered methods then use this record to isolate those proofs that are actually affected by changes to specifications. Often, only parts of these affected proofs are invalid. SID therefore has additional mechanisms for retaining still-valid subproofs.

A primary goal was to formulate the methodology in a general enough way to enable it to be easily adapted to a class of Pascal-like languages and their related proof methods. Insofar as its adaption for use in SID, this goal was realized. The main algorithms remained essentially intact with only a few auxiliary functions rewritten.

The main adjustments are as follows. When changes to specifications affect proofs, MarkToCheckVCs uses the data-base record of where specifications were actually used to directly identify the affected proofs. ExternalSpecCallsTo additionally detects recursive calls so that changes to headers will be handled properly. Before generating VCs, IsPathConsistent ensures only that entry specifications of called programs are semantically consistent.

# CHAPTER 5

# DESIGNING INCREMENTAL SYSTEMS

Previous chapters illustrated how SID responds to changes and detailed the methodology it empoloys. This chapter considers incremental systems in general, discussing design considerations, proposing a general framework for solving some key design problems, and illustrating some open problems.

## 5.1 Issues and Tradeoffs

Incremental systems respond to changes by ensuring that the final problem solution is consistent and by keeping intact as much still-valid work as practical. Both the user and system perspective on how this happens is important. From the user's viewpoint, the system keeps intact still-valid work without redoing previous work. In actuality, however, a limited amount of reprocessing may be desireable.

There is a spectrum of ways in which individual components can keep intact still-valid work. At one end of the spectrum is the most straightforward way to respond to changes -- simply redo everything, then ferret out what did not need to be redone by comparing new results with previous results, then bring to the attention of the user only work that really needs to be done. Although this "batch" approach conveys an incremental view to the user, it is often too inefficient. The approach at the other end of the spectrum is to isolate the exact impact of changes, and not redo any still-valid previous work. Since nothing is redone, the user sees nothing being redone. This too can be highly inefficient since the component may be doing as much work (or more) figuring out how to keep from redoing work as it would take to redo it. In short, the amount of effort spent isolating the exact impact of changes is inversely related to the amount of effort spent redoing previous work.

What is needed is to adopt a strategy that lies somewhere between these endpoints. The ultimate goal, of course, is to find a point at which the total amount of effort spent determining the impact of changes and redoing previous work is always minimized. The idea here is to localize the effects of changes to a single unit of information (or to a group of such units), redo everything in that locality when practical, then separate out and show to the user only previous work affected by the change. These units of information are pre-defined and are called the *grain* of the component.

Components also need to be studied collectively. The main reasons are to decide how they interface and how they need to interact to resolve inconsistencies. The interface problem, along with the intimately related one of determining the ideal grain for each component, is illustrated by considering a partially-ordered set of components arranged such that component A "comes before" components B and C. This ordering indicates that B and C depend on the results of A (e.g., a compiler and VC generator depend on the results of editing operations). The grains used by B and C impact the kind of grain that can be used by A. The way these grains interrelate determines how easy it is to interface A with B and with C. Carefully choosing the grain of A so that it is easy to extract from its output what B and C need to know about their grains makes the interfaces simple. Grain incompatibilities make interfaces difficult if not impractical.

In summary, four major interrelated questions have been raised:

- How does each component determine the effects of changes?
- How do components interact to resolve inconsistencies?
- What is the appropriate grain for each component?
- How do components fit together?

For answering the first two questions, a framework that is general enough to be adapted to different kinds of systems by filling in specific details as necessary is proposed below. Then, the last two open questions are explored.

## 5.2 General Theory

To produce a system that automatically determines the effects of changes, the system designer must fill in the following outline with task-specific details. Each component views its problem as being represented by a network of nodes, which correspond to the grain of the component, and a relation between nodes. The designer must define the nodes for each component (let $node_{c,i}$ be an arbitrary node for component c) and the relationship between them (let $uses_c$ be the relation). Also, a predicate is required for deciding whether an externally-visible part of $node_{c,i}$ $uses_c$ $node_{c,j}$ (let $UsesInExternalPart_c(node_{c,i},node_{c,j})$ be this predicate). Chapter 4 shows one way to define these items for SID. After the details are supplied, the algorithm below responds to changes by transforming the current network to another that embodies the change and whatever else is needed to compensate for it. If $node_{c,i}$ is changed, the necessary transformations are made by the following steps:

1. ***Determine local impact.*** Mark $node_{c,i}$.

2. ***Determine global impact.*** If an externally-visible part of $node_{c,i}$ is changed, mark all of its users.† These users are marked recursively by:

```
MarkUsers(node_c,i) =
    for each node_c,j such that node_c,j uses_c node_c,i
        (mark node_c,j;
         if UsesInExternalPart_c(node_c,j,node_c,i) then
             MarkUsers(node_c,j) )
```

3. ***Perform updating.*** Recompute $uses_c$ relation.

4. ***Determine effects on next component.*** If c is not the "last" component in the system, repeat these steps for each of the "next" component's nodes that come from†† the marked nodes of c.

The marking of a node indicates that the associated component must be (re)applied to it. Next, it will be shown how these markings place constraints on what can be done.

The problem of resolving inconsistencies is equivalent to deciding when a component can consistently be applied. Before applying component c to $node_{c,i}$, the following is performed:

1. ***Start at the beginning.*** Trace $node_{c,i}$ back to a node, or set of nodes, for the "first" component.

2. ***Ensure global consistency.*** Each traced node, from the first component to the one immediately preceding c in the ordering, must be unmarked.

3. ***Ensure local consistency.*** If $node_{c,i}$ is marked, all paths from $node_{c,i}$ must be unmarked.

In step 3, if $node_{c,i}$ is unmarked, the operation can be applied without any local checking because other marked nodes cannot affect it.

---

† Nodes that are added (as opposed to changed) can be handled so that they are not externally visible, causing this step to be skipped.

†† Doing this mapping, which is in general one-to-many, is part of the grain/interface problem.

Interestingly, nodes are traced "backward" through the component ordering here in contrast to "forward" as in the other algorithm. These two requirements underscore the importance of the grain/interface problem and how critical an efficient solution is.

Chapter 4 shows one way to fill in the details of these two outlines for SID. It defines the necessary items (including a grain, a uses relation, and an "externally-visible" predicate) for each individual component and shows how to interleave them.

## 5.3 Future Research

### Grains and Tools

The key observation to be brought out is that different tools typically have different grains. Several different kinds of components are considered and a possible grain for each is suggested. Even though components are discussed separately, remember that they must be viewed collectively when designing a system.

***VC generator.*** Suppose this component works by first dividing a program into paths, then generating VCs for each path. Whenever a program is changed, it needs to know if any new VCs need to be generated. In particular, it needs to know what paths, rather than what syntactic constructs (like expressions or statements) within paths, are changed. Hence, a convenient grain is a program path.

***Optimizing compiler.*** When compiling a for-loop of the form

```
for i from 1 to n do
   . . .
x←k*n;
   . . .
```

the statement "x←k*n" is properly optimized out of the loop. If, for example, the "n" in this statement is changed to an "i", the compiler is not so concerned about knowing that "n" is now "i", but about knowing that it needs to recompile the whole for-loop. Thus, an optimized unit is an obvious grain for this component. An example of an incremental compiling scheme is in Mitchell [70].

***Editor.*** A useful grain for an editor is an operation. Having this as the actual grain enables it to have three logical grains -- an operation, a sequence of operations, and an editing session. For example, all these are useful for providing a flexible do-undo feature that reflects

both program development and verification concerns. INTERLISP [Teitelman 75] uses this feature effectively for program development.

*Equivalence-preserving transformer.* This component takes an abstract program and converts it into a more concrete one by successive transformations. These transformations are selected by the programmer, then applied automatically. Since transformations are equivalence-preserving, the program produced is guaranteed to be a valid implementation of the abstract program. Such schemes (e.g., Balzer, et al. [76], Gerhart [75], and Standish [76]) typically operate on program segments.

As a simple example of what an incremental transformer does, consider the following sequence of statements:

```
for i from 1 to n do B[i]←i+1;
for j from 1 to n do C[j]←B[j]+D[j];
```

These two for-loops can be combined into a single one, viz.,

```
for i from 1 to n do
    begin B[i]←i+1; C[i]←B[i]+D[i] end;
```

The transformer must, however, account for the fact that a change to the original segment may invalidate this transformation. Thus, it needs to keep program segment-transformed segment pairs, making it convenient to deal with changes at the segment level.

## Fitting the Pieces Together

Interfacing a set of components requires establishing a partial ordering of components and being able to map between nodes of different components. The latter is particularly difficult.

One of the key decisions is deciding how changes will be introduced into the system. To isolate exact changes, an editor is required as the "first" component in the system. The reason for this requirement is that the alternative of using a "pattern match" approach, while effective in some situations (as shown in SID), is highly unsatisfactory in isolating exact changes. It may require excessive processing to isolate exact changes (e.g., equivalence-preserving operations like uniformly renaming a variable are difficult to identify) in part because the sequence of editing operations is lost.

The ideal editor not only executes commands, but also is aware of the context in which they are executed. The impact of executing an editorial command is determined by

mapping the editor's context to the "equivalent" context for other components. This is done by one of the algorithms in Section 5.2.

Arriving at specific guidelines for designing these mappings, as well as for choosing grains, are important areas for future research.

# REFERENCES

Ambler, A. L., D. I. Good, J. C. Browne, W. F. Burger, R. M. Cohen, C. G. Hoch, and R. E. Wells [77]. Gypsy: A language for specification and implementation of verifiable programs, *Proceedings of an ACM Conference on Language Design for Reliable Software, SIGPLAN Notices*, 12, 3, March 1977, 1-10.

Balzer, R., N. Goldman, and D. Wile [76]. On the transformational implementation approach to programming, *Proceedings of Second International Conference on Reliable Software*, ACM and IEEE, October 1976, 337-344.

Bledsoe, W. W. [75]. A new method for proving certain Presburger formulas, *Advance Papers of the Fourth International Joint Conference on Artificial Intelligence*, 1975, 15-21.

Bledsoe, W. W., and P. Bruell [73]. A man-machine theorem-proving system, *Advance Papers of Third International Joint Conference on Artificial Intelligence*, 1973, 56-65. Also *Artificial Intelligence*, 5, 1, Spring 1974, 51-72.

Bledsoe, W. W., and M. Tyson [75]. The UT interactive prover, University of Texas at Austin Mathematics Department Memo ATP-17, May 1975.

Boyer, R. S., and J S. Moore [75]. Proving theorems about Lisp functions, *J. ACM*, 22, 1, January 1975, 129-144.

Boyer, R. S., and J S. Moore [77]. A lemma driven automatic theorem prover for recursive function theory, *Proceedings of Fifth International Joint Conference on Artificial Intellegence*, August 1977, 511-519.

Burger, W. [74]. BOBSW - A parser generator, University of Texas at Austin Report SESLTR-7, December 1974.

Carter, W. C., H. A. Ellozy, W. H. Joyner, Jr., G. B. Leeman, Jr. [77]. Techniques for microprogram validation, IBM T. J. Watson Research Center Report RC 6361, September 1977.

Crocker, S. D. [77]. *State deltas: A formalism for representing segments of computation*, USC Information Sciences Institute Report ISI/RR-77-61, September 1977.

Dahl, O.-J. and C. A. R. Hoare [72]. Hierarchical program structures, in *Structured Programming* (O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare), Academic Press, 1972, 175-200.

Deutsch, L. P. [73]. An interactive program verifier, Ph. D. thesis, University of California-Berkeley, 1973. Also Xerox Palo Alto Research Center Report CSL-73-1, May 1973.

Elspas, B., K. N. Levitt, and R. J. Waldinger [73]. An interactive system for the verification of computer programs, Stanford Research Institute Project 1891 Final Report, September 1973.

Ernst, G. W., and R. J. Hookway [75]. Formulating inductive assertions for program verification, Case Western Reserve University Computing Center Report No. 1165, March 1975.

Floyd, R. W. [67]. Assigning meanings to programs, *Proceedings of a Symposium in Applied Mathematics*, Vol. 19, J. T. Schwartz, ed., American Mathematical Society, 1967, 19-32.

Gerhart, S. L. [75]. Knowledge about programs: A model and a case study, ICRS [75], 88-95.

Good, D. I. [70]. Toward a man-machine system for proving program correctness, Ph. D. thesis, University of Wisconsin, 1970. Also University of Texas at Austin Computation Center Report TSN-11.

Good, D. I., ed., [77]. Constructing verifiably reliable and secure communications processing systems, Institute for Computing Science and Computer Applications Report ICSCA-CMP-6, The University of Texas at Austin, January 1977.

Good, D. I., R. L. London, and W. W. Bledsoe [75]. An interactive program verification system, ICRS [75], 482-492. Also *IEEE Transactions on Software Engineering*, SE-1, 1, March 1975, 59-67.

Hearn, A. C. [71]. Reduce 2: A system and language for algebraic manipulation, *Proceedings of Second Symposium on Symbolic and Algebraic Manipulation*, ACM, 1971, 128-133. Also Reduce 2 user's manual, University of Utah UCP-19, second edition, 1974.

von Henke, F. W., and D. C. Luckham [75]. A methodology for verifying programs, ICRS [75], 156-163. Also Automatic program verification III: A methodology for verifying programs, Stanford University Artificial Intelligence Laboratory Memo AIM-256, December 1974.

Hoare, C. A. R. [71]. Proof of a program: FIND, *Comm. ACM*, 14, 1, January 1971, 39-45.

Hookway, R. J. [76]. A program verification system, M.S. thesis, Case Western Reserve University, 1976. Also Case Western Reserve Computing Center Report No. 1171, January 1976.

ICRS [75]. *Proceedings of International Conference on Reliable Software*, April 1975. Also *SIGPLAN Notices*, 10, 6, June 1975.

Igarashi, S., R. L. London, and D. C. Luckham [73]. Automatic program verification I: A logical basis and its implementation, Stanford University Artificial Intelligence Laboratory Memo AIM-200, May 1973. Also *Acta Informatica*, 4, 2, 1975, 145-182.

Karp, R. A., and D. C. Luckham [76]. Verification of fairness in an implementation of monitors. *Proceedings of International Conference on Reliable Software*, October 1976, 40-46.

King, J. C. [69]. A program verifier, Ph. D. thesis, Carnegie-Mellon University, 1969.

King, J. C. [71]. Proving programs to be correct, *IEEE Transactions on Computers*, C-20, 11, November 1971, 1331-1336.

Liskov, B. [75]. A note on CLU, Massachusetts Institute of Technology MAC-TR, June 1975.

Luckham, D. C., and N. Suzuki [75]. Automatic program verification IV: Proof of termination within a weak logic of programs, Stanford University Artificial Intelligence Laboratory Memo AIM-269, October 1975.

Luckham, D. C., and N. Suzuki [76]. Verification oriented proof rules for arrays, records, and pointers, Stanford University Artificial Intelligence Laboratory Memo 278, April 1976.

Marmier, E. [75]. Automatic verification of Pascal programs, Ph. D. thesis, Swiss Federal Institute of Technology (ETH), Zurich, 1975.

Milner, R. [72]. Implementation and applications of Scott's logic for computable functions, *SIGPLAN Notices*, 7, 1, January 1972. Also *SIGACT News*, 14, January 1972, 1-6.

Mitchell, J. G. [70]. The design and construction of flexible and efficient interactive programming systems, Ph.D. thesis, Carnegie-Mellon University, June 1970.

Moore, J S. [73]. Computational logic: Structure sharing and proving program properties, Ph. D. thesis, University of Edinburgh, 1973.

Moore, J S. [75]. Introducing iteration into the pure Lisp theorem prover, Xerox Palo Alto Research Center Report CSL-74-3, December 1974 (revised March 1975).

Musser, D. R. [77]. A data type verification system based on rewrite rules, *Proceedings of Sixth Texas Conference on Computing Systems*, November 1977, (1A) 22-30.

Rulifson, J. F., J. A. Derksen, R. J. Waldinger [72]. QA4: A procedural calculus for intuitive reasoning, Stanford Research Institute Project 8721 Final Report, November 1972.

Schorre, V. [76]. A program verifier with assertions in terms of abstract data, *Proceedings of the Symposium on Computer Software Engineering*, Polytechnique Press, 1976, 267-280.

Standish, T. D., D. Harriman, D. Kibler, and J. Neighbors [76]. *The Irvine Program Transformation Catalog*, Department of Information and Computer Sciences, University of California at Irvine, January 1976.

Suzuki, N. [75]. Verifying programs by algebraic and logical reduction, ICRS [75], 473-481. Also Automatic program verification II: Verifying programs by algebraic and logical reduction, Stanford University Artificial Intelligence Laboratory Memo AIM-255, December 1974.

Teitelman, W. [75]. *INTERLISP Reference Manual*, XEROX Palo Alto Research Center, 1975.

Topor, R. W. [75]. Interactive program verification using virtual programs, Ph. D. thesis, University of Edinburgh, 1975.

Waldinger, R. J., and K. N. Levitt [74]. Reasoning about programs, *Artificial Intelligence*, 5, 3, Fall 1974, 235-316. Also *Conference Record of ACM Symposium on Principles of Programming Languages*, 1973, 169-182.

Wulf, W. A., R. L. London, M. Shaw [76]. *Abstraction and verification in Alphard: Introduction to language and methodology*, USC Information Sciences Institute Technical Report ISI/RR-76-46, June 1976.

Yonke, M. D. [75]. *A knowledgeable, language-independent system for program construction and modification*, USC Information Sciences Institute Report ISI/RR-75-42, October 1975.

Yonke, M. D. [76]. The XIVUS environment, USC Information Sciences Institute working paper no. 1, April 1976.